FILED
2012 OCT 31 P 3:45

Jeff D. Friedman (173886)
HAGENS BERMAN SOBOL SHAPIRO LLP
715 Hearst Avenue, Suite 202
Berkeley, CA 94710
Telephone: (510) 725-3000
Facsimile: (510) 725-3001
jefff@hbsslaw.com

Steve W. Berman (*Pro Hac Vice* Application to be filed)
Mark S. Carlson (*Pro Hac Vice* Application to be filed)
HAGENS BERMAN SOBOL SHAPIRO LLP
1918 Eighth Avenue, Suite 3300
Seattle, WA 98101
Telephone: (206) 623-7292
Facsimile: (206) 623-0594
steve@hbsslaw.com
markc@hbsslaw.com

Attorneys for Plaintiff
Thought, Inc.

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

LB

THOUGHT, INC., a California corporation, Plaintiff, v. ORACLE CORPORATION, a Delaware corporation; ORACLE AMERICA, INC, a Delaware corporation; and ORACLE INTERNATIONAL CORPORATION, a California corporation Defendants.	C No. 12 5601 COMPLAINT FOR PATENT INFRINGEMENT **DEMAND FOR JURY TRIAL**

Plaintiff Thought, Inc. ("Thought"), for its complaint against defendants Oracle Corporation, Oracle America, Inc., and Oracle International Corporation (collectively "Oracle"), alleges as follows:

I. NATURE OF THE ACTION

1. This is an action under the patent laws of the United States of America, 35 U.S.C. §§ 1, *et seq.*, for infringement of patents assigned to Thought.

II. THE PARTIES

2. Thought is a corporation, organized and existing under the laws of the State of California. Thought's principal place of business is 5 Third Street, Suite 1030, San Francisco California 94103.

3. On information and belief, Oracle Corporation is a corporation organized and existing under the laws of the State of Delaware, with its principal place of business at 500 Oracle Parkway, Redwood City, CA 94065.

4. On information and belief, Oracle America, Inc. is a corporation organized and existing under the laws of the State of Delaware, with its principal place of business at 500 Oracle Parkway, Redwood City, CA 94065.

5. On information and belief, Oracle International Corporation is a corporation organized and existing under the laws of the State of California, with its principal place of business at 500 Oracle Parkway, Redwood City, CA 94065.

III. JURISDICTION AND VENUE

6. This Court has subject matter jurisdiction over this action pursuant to 28 U.S.C. §§ 1331 and 1338, in that this is a civil action for patent infringement arising under the Patent Laws of the United States, Title 35, United States Code.

7. Venue is proper in this judicial district under 28 U.S.C. §§ 1391(b)(1) and (c)(2), and 1400(b).

8. This Court has personal jurisdiction over Oracle because Oracle International is a California corporation, Oracle has its principal place of business in this judicial district, and Oracle

conducts systematic and continuous business in California within this judicial district, and has committed acts of infringement in California and within this judicial district.

IV. INTRADISTRICT ASSIGNMENT

9. This is an Intellectual Property Action to be assigned on a district-wide basis under Civil L.R. 3-2(c) and 3-5.

V. BACKGROUND OF THE INVENTION AND THOUGHT

10. Thought was founded by computer engineer Ward Mullins in 1993 to deliver object-oriented programming solutions to corporate clients. Following the release of the Java programming language by Sun Microsystems in 1995, Ward Mullins and others at Thought investigated the new language and determined that it provided an effective platform to develop mapping technologies between corporate applications and databases, thereby replacing the cumbersome, inefficient, labor-intensive, and generally *ad hoc* solutions used previously.

11. Using Java, Thought invented a middleware mapping layer for saving object and table information and greatly simplified the task of persisting data. Thought's dynamic object to relational mapping layer made it possible to keep the database information in a map and not as programming code in the Java object, meaning no bytecode or extension code generation or annotation was needed either at design or runtime in order to persist the data of a Java object.

12. Thought incorporated its innovative technologies into its CocoBase® product introduced in January 1997. CocoBase® was licensed widely to numerous customers and partners over the years.

13. Thought continued to develop its CocoBase® product, and developed further innovations related to its object to relational mapping technologies, including features such as object navigation, object persistence modeling, dynamic object-based querying, and advanced object data caching.

VI. THE PATENTS-IN-SUIT

14. Thought is the assignee and owner of U.S. Patent No. 5,857,197 (the "'197 Patent"), entitled *System and Method for Accessing Data Stores as Objects*. The '197 Patent was duly and legally issued by the United States Patent and Trademark Office on January 5, 1999 and generally

discloses and claims systems and methods to perform object to relational database mapping by associating objects in an object application with corresponding data that persists over time in a data store. The '197 Patent is attached to this Complaint as Exhibit A.

15. Thought is the assignee and owner of U.S. Patent No. 6,985,912 (the "'912 Patent"), entitled *Dynamic Object-Driven Database Manipulation and Mapping System Having a Simple Global Interface and an Optional Multiple User Need Only Caching System with Disable and Notify Feature*. The '912 Patent was duly and legally issued by the United States Patent and Trademark Office on January 10, 2006 and generally discloses and claims systems and methods to cache certain objects and data. The '912 Patent is attached to this Complaint as Exhibit B.

16. Thought is the assignee and owner of U.S. Patent No. 6,999,956 (the "'956 Patent"), entitled *Dynamic Object-Driven Database Manipulation and Mapping System*. The '956 Patent was duly and legally issued by the United States Patent and Trademark Office on February 14, 2006 and generally discloses and claims systems and methods to create and edit object mapping files in an object to relational database mapping system. The '956 Patent is attached to this Complaint as Exhibit C.

17. Thought is the assignee and owner of U.S. Patent No. 7,043,481 (the "'481 Patent"), entitled *System, Method, and Software for Creating, Maintaining, Navigating, or Manipulating Complex Data Objects and Their Data Relationships*. The '956 Patent was duly and legally issued by the United States Patent and Trademark Office on May 9, 2006 and generally discloses and claims systems and methods to create, maintain, and access complex data objects as a complex data object graph model. The '481 Patent is attached to this Complaint as Exhibit D.

18. Thought is the assignee and owner of U.S. Patent No. 7,103,600 (the "'600 Patent"), entitled *Displayable Presentation Page and SQL Searchable Relational Data Source Implementation of a System, Method, and Software for Creating or Maintaining Distributed Transparent Persistence of Complex Data Objects and Their Data Relationships*. The '600 Patent was duly and legally issued by the United States Patent and Trademark Office on September 5, 2006 and generally discloses and claims systems and methods for distributed transparent

persistence of complex data objects and associated data stores. The '600 Patent is attached to this Complaint as Exhibit E.

19. Thought is the assignee and owner of U.S. Patent No. 7,149,730 (the "'730 Patent"), entitled *Dynamic Class Inheritance and Distributed Caching with Object Relational Mapping and Cartesian Model Support in a Database Manipulation and Mapping System*. The '730 Patent was duly and legally issued by the United States Patent and Trademark Office on December 12, 2006 and generally discloses and claims systems and methods for object to relational mapping and caching. The '730 Patent is attached to this Complaint as Exhibit F.

20. Thought is the assignee and owner of U.S. Patent No. 7,176,862 (the "'862 Patent"), entitled *Session Bean Implementation of a System, Method, and Software for Creating or Maintaining Distributed Transparent Persistence of Complex Data Objects and Their Data Relationships*. The '862 Patent was duly and legally issued by the United States Patent and Trademark Office on January 23, 2007 and generally discloses and claims systems and methods for creating or maintaining distributed transparent persistence of complex data objects and associated data stores.

VII. BACKGROUND ON ORACLE

21. In its publicly-filed 2011 Form 10K Oracle summarizes its business as follows:

> Oracle is the world's largest provider of enterprise software and a leading provider of computer hardware products and services. Our software, hardware systems, and services businesses develop, manufacture, market, host and support database and middleware software, applications software, and hardware systems, with the latter consisting primarily of computer server and storage products. Our businesses provide products and services that are built upon industry standards, are engineered to work together or independently within existing customer information technology (IT) environments, and run securely on a wide range of customer IT environments, including cloud computing environments.

22. On January 16, 2008, Oracle announced the acquisition of all outstanding shares of BEA Systems, Inc. in a deal valued at approximately 8.5 billion dollars. Oracle completed the acquisition on or about April 29, 2008. At the time of the acquisition, BEA's flagship product was the Weblogic application server. Oracle CEO Larry Ellison stated that "The addition of BEA

products and technology will significantly enhance and extend Oracle's Fusion middleware software suite."

23. On April 20, 2009, Oracle announced the acquisition of Sun Microsystems, Inc. in a deal valued at approximately 7.4 billion dollars. Oracle completed the acquisition on or about January 27, 2010. At the time of the acquisition Sun's flagship software assets were the Java programming language and Solaris operating system. In the acquisition announcement, Oracle noted that the Oracle Fusion Middleware was "Oracle's fastest growing business, [and] is built on top of Sun's Java language and software." Oracle CEO Larry Ellison stated:

> "Oracle will be the only company that can engineer an integrated system – applications to disk – where all the pieces fit and work together so customers do not have to do it themselves. Our customers benefit as their systems integration costs go down while system performance, reliability and security go up."

24. Oracle continues to deploy the Oracle Weblogic Server and boasts that it is the "foundation" of Oracle Fusion Middleware software and is "compliant with the Java EE specification. Oracle WebLogic Server incorporates clustering and caching technology, which increases application reliability, performance, security and scalability." On information and belief, accused versions of Oracle Weblogic Server are compliant with either or both Java EE 5 and Java EE 6.

25. Oracle also offers database and application software, hardware, application development tools, and hosting and other cloud services that rely on its middleware including Oracle Weblogic Server.

VIII. ORACLE'S KNOWLEDGE OF THOUGHT'S PATENT PORTFOLIO

26. Each version of Thought's CocoBase® product was marked with the '197 Patent following its issuance by the United States Patent and Trademark Office on January 5, 1999.

27. Oracle has cited Thought's '197 Patent as prior art on the face of U.S. Patents 7,599,948 and 8,145,685 assigned to Oracle International Corporation.

28. Oracle has cited Thought's '956 Patent as prior art on the face of U.S. Patents 7,882,132, 7,904,487, 7,853,573, 7,809,763, 7,801,856, 7,689,580, 7,630,974,7,613,794,

7,526,490, and 7,512,585 assigned to Oracle International Corporation as well as 7,805,507 and 7,711,625 assigned to Oracle America, Inc.

29. Oracle has cited the '912 Patent as prior art on the face of U.S. Patents 7,904,487, 7,882,132, 7,809,763, 7,630,974, 7,613,794, 7,512,585 assigned to Oracle International Corporation.

30. Between October 2, 1997 and February 25, 2004, twenty eight (28) requests for an evaluation version of Thought's CocoBase® software were received from email addresses associated with oracle.com (14), us.oracle.com (2), and dk.oracle.com (12 total, 1 unique). Each request generated a responsive email from Thought that included a link to download an evaluation version of the CocoBase® software. Each version downloaded after January 5, 1999 included information stating that the software was covered by the claims of the '197 Patent.

31. Between September 30, 1997 and June 2, 2003, one hundred and eighteen (118) requests for an evaluation version of Thought's CocoBase® software were received from email addresses associated with sun.com (63 total, 58 unique), sweden.sun.com (2), singapore.sun.com (3 total, 2 unique), east.sun.com (5 total, 4 unique), west.sun.com (1), central.sun.com (1), italy.sun.com (1), canada.sun.com (1), germany.sun.com (1), austria.sun.com (2 total, 1 unique), india.sun.com (1), aus.sun.com (1), and belgium.sun.com (36 total, 1 unique). Each request generated a responsive email from Thought that included a link to download an evaluation version of the CocoBase® software. Each version downloaded after January 5, 1999 included information stating that the software was covered by the claims of the '197 Patent. This notice and actual knowledge of Thought's CocoBase® software and patent portfolio passed to Oracle as a result of Oracle's acquisition of Sun Microsystems, Inc. in 2010.

32. In an email dated May 2, 2002 to Ted Farrell, currently Chief Architect and Senior Vice President Tools and Middleware at Oracle, Ward Mullins of Thought identified certain technologies covered by Thought's current and then-pending patent rights so that Oracle would not "accidentally violate [Thought's] intellectual property." The email specifically called out the following technologies:

- Repository based O/R (object relational) mapping, specifically referencing the '197 Patent;
- Navigation of the object model based upon a repository, generally referencing Thought's then pending patent applications;
- Dynamic persistence, for example by using a session bean or an entity bean, generally referencing Thought's then pending patent applications;
- Code generation based upon a repository of metadata related to O/R (object relational) mapping, generally referencing Thought's then pending patent applications.

33. In an email dated August 4, 2003 to Craig Russell of Sun Microsystems, currently an Architect at Oracle, Dan Wilson of Thought declined an invitation from Russell to participate in a conference to develop a new Java Data Objects specification, called JDO2. Wilson declined to participate due to potential adverse effects on Thought's patent and pending patent rights. Wilson did suggest that Russell and the Sun technology licensing committee review the CocoBase® solution to further understand Thought's views of the proper changes to the Java Data Objects specification. Wilson also noted that if JDO2 incorporated Thought's patent-protected technology, Thought would enter into negotiations with Sun and others to discuss an appropriate license fee.

34. Craig Russell was specification lead on JDO2 and, on information and belief, was involved with JSR 220 and JSR 317 Expert Groups that merged JDO2 into the Java Persistence API.

35. Thought's CocoBase® architecture and patented technology were incorporated into the JSR 220: Enterprise JavaBeansTM, version 3.0 specification of the Java Persistence API in May, 2006. Thought's patented and then-patent pending technology was further adopted by the JSR 317 Expert Group with Java Persistence API 2.0. Thought's patented technology was adopted without Thought's consent.

36. Oracle also has notice of Thought's patent portfolio through service of this Complaint.

IX. COUNT I

INFRINGEMENT OF THE '197 PATENT

37. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-36 of this Complaint as if fully set forth herein.

38. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '197 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. These infringing Oracle products include products that embody the EJB 3.0 and later standards and Java Persistence API. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.0 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 10.1.3.5 and later, all versions of Sun/Oracle Glassfish Server, Exalogic Elastic Cloud versions X2-2 and X3-2, and Oracle JDeveloper 10.1.3.1 and later.

39. Oracles's infringement of the '197 Patent has been on a massive scale, and has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

40. As a result of Oracle's infringement of the '197 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

41. As a result of Oracle's infringement of the '197 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

X. COUNT II

INFRINGEMENT OF THE '912 PATENT

42. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-41 of this Complaint as if fully set forth herein.

43. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '912 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. These infringing Oracle products include products that embody Java Persistence API 2.0 and later. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.3 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 11.1.1.3 and later, Sun/Oracle Glassfish Server version 3.0 and later, Oracle Coherence version 3.5 and later, and Exalogic Elastic Cloud versions X2-2 and X3-2.

44. Oracles's infringement of the '912 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

45. As a result of Oracle's infringement of the '912 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

46. As a result of Oracle's infringement of the '912 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XI. COUNT III

INFRINGEMENT OF THE '956 PATENT

47. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-46 of this Complaint as if fully set forth herein.

48. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '956 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet

all of the limitations of the infringed claims. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.0 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 10.1.3.5 and later, all versions of Sun/Oracle Glassfish Server, Exalogic Elastic Cloud versions X2-2 and X3-2, Oracle JDeveloper 10.1.3.1 and later, and NetBeans IDE 6.01 and later.

49. Oracles's infringement of the '912 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

50. As a result of Oracle's infringement of the '912 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

51. As a result of Oracle's infringement of the '912 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XII. COUNT IV

INFRINGEMENT OF THE '481 PATENT

52. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-51 of this Complaint as if fully set forth herein.

53. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '481 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.0 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 10.1.3.5 and later, all versions of Sun/Oracle Glassfish Server, Exalogic Elastic Cloud versions X2-2 and X3-2, Oracle JDeveloper 10.1.3.1 and later, and NetBeans IDE 6.01 and later.

54. Oracles's infringement of the '481 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

55. As a result of Oracle's infringement of the '481 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

56. As a result of Oracle's infringement of the '481 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XIII. COUNT V

INFRINGEMENT OF THE '600 PATENT

57. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-56 of this Complaint as if fully set forth herein.

58. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '600 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. These infringing Oracle products include products that embody the EJB 3.0 and later standards and Java Persistence API. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.3 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 11.1.1.3 and later, Sun/Oracle Glassfish Server version 3.0 and later, and Exalogic Elastic Cloud versions X2-2 and X3-2.

59. Oracles's infringement of the '600 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making,

using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

60. As a result of Oracle's infringement of the '600 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

61. As a result of Oracle's infringement of the '600 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XIV. COUNT VI

INFRINGEMENT OF THE '730 PATENT

62. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-61 of this Complaint as if fully set forth herein.

63. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '730 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. These infringing Oracle products include products that embody the EJB 3.0 and later standards and Java Persistence API. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.3 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 11.1.1.3 and later, Sun/Oracle Glassfish Server version 3.0 and later, and Exalogic Elastic Cloud versions X2-2 and X3-2.

64. Oracles's infringement of the '730 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

65. As a result of Oracle's infringement of the '730 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

66. As a result of Oracle's infringement of the '730 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XV. COUNT VII

INFRINGEMENT OF THE '862 PATENT

67. Thought re-alleges and incorporates herein by reference the allegations stated in paragraphs 1-66 of this Complaint as if fully set forth herein.

68. Oracle has been and continues to directly and/or indirectly infringe (literally or under the doctrine of equivalents) one or more claims of the '862 Patent by making, using, offering for sale, selling and/or importing into the United States products that either alone, or in combination with other products for which they are intended to be used, include elements that meet all of the limitations of the infringed claims. These infringing Oracle products include products that embody the EJB 3.0 and later standards and Java Persistence API. Accused products include, but are not necessarily limited to, the following: Oracle Weblogic Server version 10.3.3 and later including Oracle Cloud Services that use Oracle Weblogic Server, Oracle TopLink version 11.1.1.3 and later, Sun/Oracle Glassfish Server version 3.0 and later, and Exalogic Elastic Cloud versions X2-2 and X3-2.

69. Oracles's infringement of the '862 Patent has been on a massive scale, and on information and belief has taken place with actual knowledge of the inventions claimed therein. On information and belief, Oracle undertook an objectively high likelihood that its actions in making, using, offering for sale, selling and/or importing into the United States the accused products constituted infringement of a valid patent and are therefore willful infringement.

70. As a result of Oracle's infringement of the '862 Patent, Thought has been and will continue to be irreparably harmed unless and until Oracle's infringement is enjoined by this Court.

71. As a result of Oracle's infringement of the '862 Patent, Thought has been and will continue to be damaged in an amount to be proved at trial, but not less than a reasonable royalty for each infringement.

XVI. PRAYER FOR RELIEF

Wherefore, Thought respectfully requests that this Court:

1. Enter a judgment in favor of Thought that Oracle has infringed one or more claims of the '197 Patent, '912 Patent, '956 Patent, '481 Patent, '600 Patent, '730 Patent, and '862 Patent;

2. Grant a permanent injunction enjoining Oracle, its officers, directors, agents, servants, affiliates, employees, successors, assigns, divisions, branches, subsidiaries, parents and all others acting in active concert therewith from infringing and/or inducing others to infringe or contribute to the infringement of the '197 Patent, '912 Patent, '956 Patent, '481 Patent, '600 Patent, '730 Patent, and '862 Patent;

3. Award Thought damages in an amount sufficient to compensate for Oracle's infringement of the '197 Patent, '912 Patent, '956 Patent, '481 Patent, '600 Patent, '730 Patent, and '862 Patent in an amount to be proved at trial, but not less than a reasonable royalty;

4. Award prejudgment interest to Thought under 35 U.S.C. § 284;

5. If supported by the evidence, award increased damages, under 35 U.S.C. § 284, in an amount not less than three times the amount of actual damages awarded to Thought;

6. If supported by the evidence, declare this case exceptional under 35 U.S.C. § 285 and award Thought reasonable attorney's fees; and

7. Grant Thought such other and further relief as this Court deems just and equitable.

//

//

//

//

//

//

XVII. DEMAND FOR JURY TRIAL

Thought hereby demands a trial by jury on all issues so triable.

DATED: October 31, 2012

HAGENS BERMAN SOBOL SHAPIRO LLP

By _____
Jeff D. Friedman (173886)

HAGENS BERMAN SOBOL SHAPIRO LLP
715 Hearst Avenue, Suite 202
Berkeley, CA 94710
Telephone: (510) 725-3000
Facsimile: (510) 725-3001
jefff@hbsslaw.com

Steve W. Berman (*Pro Hac Vice* Application to be Filed)
Mark S. Carlson (*Pro Hac Vice* Application to be Filed)
HAGENS BERMAN SOBOL SHAPIRO LLP
1918 Eighth Avenue, Suite 3300
Seattle, WA 98101
Telephone: (206) 623-7292
Facsimile: (206) 623-0594
steve@hbsslaw.com
markc@hbsslaw.com

Attorneys for Plaintiff
Thought, Inc.

EXHIBIT A

US005857197A

United States Patent [19]

Mullins

[11] Patent Number: 5,857,197

[45] Date of Patent: Jan. 5, 1999

[54] **SYSTEM AND METHOD FOR ACCESSING DATA STORES AS OBJECTS**

[75] Inventor: **Ward Mullins**, San Francisco, Calif.

[73] Assignee: **Thought Inc.**, San Francisco, Calif.

[21] Appl. No.: **822,254**

[22] Filed: **Mar. 20, 1997**

[51] **Int. Cl.**[6] **G06F 17/30**

[52] **U.S. Cl.** **707/103**; 707/102

[58] **Field of Search** 707/103, 102

[56] **References Cited**

U.S. PATENT DOCUMENTS

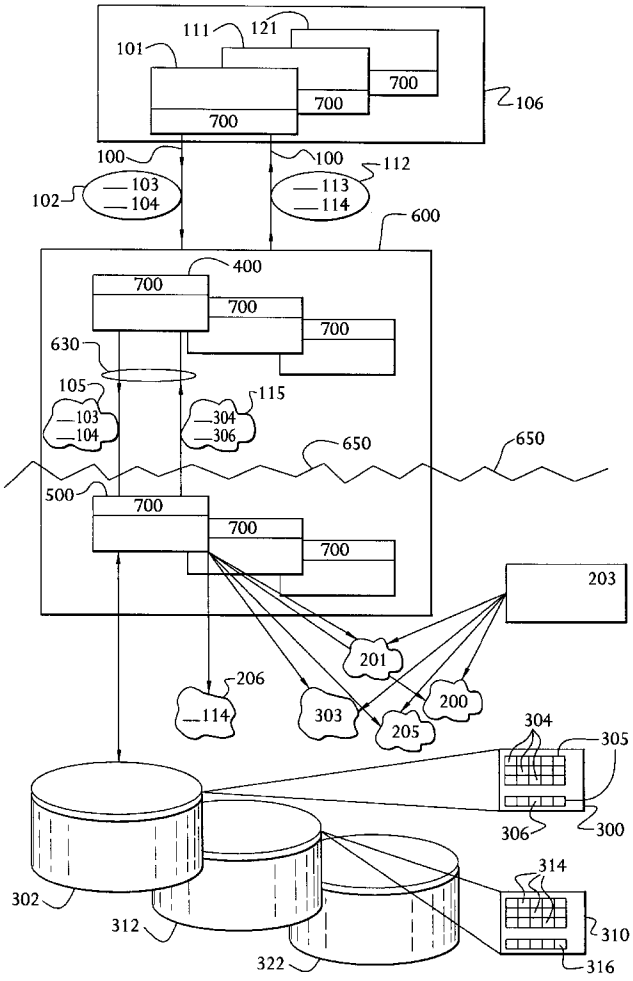
5,212,787	5/1993	Baker et al.	395/600
5,257,366	10/1993	Adair et al.	395/600
5,418,950	5/1995	Li et al.	395/600
5,542,078	7/1996	Martel et al.	395/600
5,717,877	2/1998	Orton et al.	395/326
5,720,007	2/1998	Hekmatpour	706/50
5,727,950	3/1998	Cook et al.	434/350
5,761,499	6/1998	Sondregger	707/10
5,761,509	6/1998	Danforth et al.	395/701
5,767,854	6/1998	Anwar	345/355
5,778,368	7/1998	Hogan et al.	707/10
5,784,553	7/1998	Kolawa et al.	395/183.14

Primary Examiner—Wayne Amsbury
Attorney, Agent, or Firm—Seidel, Gonda, Lavorgna & Monaco, PC

[57] **ABSTRACT**

A system and a method for accessing a data store as objects from an object application. The accessed data store could be either an object data store or a non-object (e.g. relational) data store. The system includes an object schema including meta data corresponding to a data store schema and an adapter abstraction layer. The adapter abstraction layer comprises a first adapter, and a second adapter. One embodiment of the system includes an object schema manager to create and maintain the object schema at run time. It comprises a dynamic, scalable, centrally managed, and secure method for accessing data stored in both object and non-object (e.g. relational) data stores, effecting a consistent interface to the data store regardless of its underlying structure, or a method of transport and level of security.

8 Claims, 1 Drawing Sheet



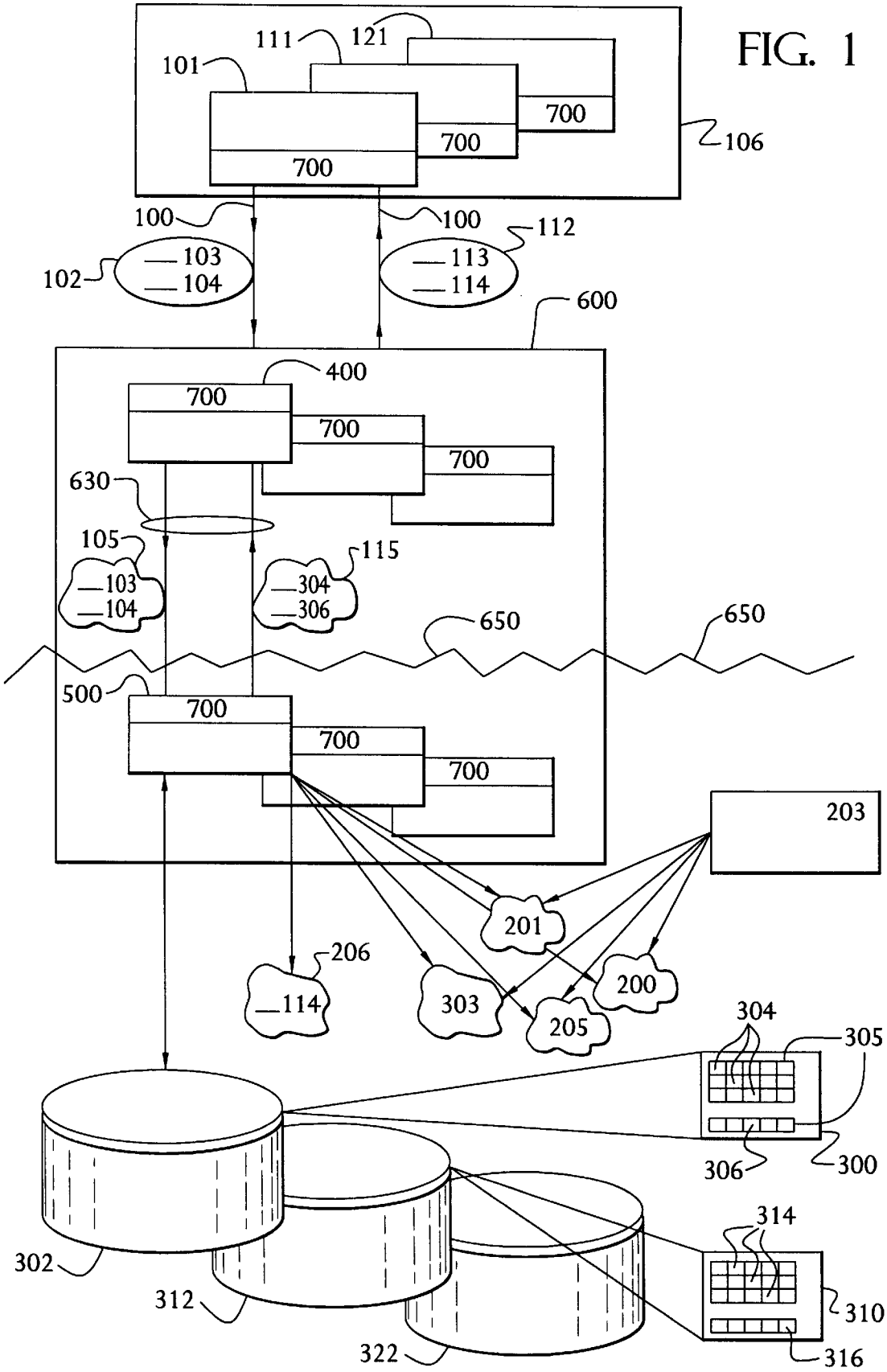


FIG. 1
101
111
121
700
700
700
106
100
100
112
102
103
104
113
114
600
400
700
700
700
630
105
115
103
104
304
306
650
650
500
700
700
700
203
201
206
114
303
200
205
304
305
306
300
302
312
314
310
322
316

SYSTEM AND METHOD FOR ACCESSING DATA STORES AS OBJECTS

FIELD OF THE INVENTION

This invention relates to a system and method for accessing a data store—which could be either an object or a non-object data store—as an object from an object application. The system includes an object schema including meta data corresponding to a data store schema and an abstraction layer. In one embodiment, the abstraction layer includes a first adapter and a second adapter. One embodiment of the invention includes an object schema manager to create and maintain the object schema as meta data, and an access adapter to access and manipulate the data store using the meta-data at run time.

BACKGROUND OF THE INVENTION

Systems for accessing data stores from object oriented languages have been used for many years. A frequent approach to accomplish access of data stores involves writing and embedding custom access code within an object application needing the access. This approach is generally limited to having the custom code access only a single relational table within a relational database or similar construct within any other data store (hereinafter collectively "data store"). Under the circumstances where a developer has control over the design and creation of a data store from its inception, it is possible to design and store meaningful information in a single table. Such design opportunities are usually rare, however.

Generally, the design goals of a properly normalized relational data store for example conflict with those of object application models. In most situations, the respective structures simply do not conveniently allow for mapping of the application object into a single relational table. This results in application objects which logically span multiple relational tables, and which require multiple translation modules in order to do useful work. Such an approach is both costly and unwieldy in terms of computing and development resources.

In those circumstances where the object can conveniently map to a single relational table, the general design philosophy prevailing in the programming industry is to embed access code in the application code itself. Because the current techniques involve the creation of custom code for each table being accessed, it is costly and time consuming to create and maintain such access applications. It means that either custom objects for access must be created for each object (such as shown in U.S. Pat. No. 5,212,787), or that the access code be embedded in the object itself, which is a common technique used in the industry. In either case, separate database specific code must be embedded in an application object. Such code specific embedding affects data store portability and code maintenance as well as significantly impacts distribution/bug-fix issues.

The recent growth in popularity and acceptance of object oriented programming languages has required a more object-oriented view of data stores. There are many different types and kinds of data stores. The need to consistently view the data stores with objects is universal. Current systems are inflexible, inconsistent and inadequate to meet this need.

The common modeling technique of the current systems is to map the underlying non-object data types directly as objects, as opposed to a more logical view of the data which may span many physical data stores or data store objects. Viewing non-object entities directly as objects makes migration from object stores to non-object stores and back again difficult and inefficient.

Further difficulties are created by mapping tools that attempt to map objects to non-object data stores. The common technique in such mapping is to 'generate' an object view which is then embedded in the object application code. Unfortunately any minor modification to the underlying non-object data store can break application code, thus creating a 'brittle client'. As used in this application, brittle client refers to a situation where a client application breaks under minor changes to the data store's physical layout (i.e. data store schema), which require no 'logical' changes in the object application logic, but which requires embedded access code to change in order to successfully use the object application. Another common approach of the prior art is to use another intermediate, or an 'object store', also commonly referred to as a gateway (see e.g. U.S. Pat. No. 5,542.078), where mapping occurs. While this is a workable solution for some, many companies do not need nor want the extra management and complexity of yet another data store system. The facilities provided for data management, integrity, and record locking are often duplicated in both systems unnecessarily, providing overly redundant and complex systems. Most gateways also suffer in various degrees from lack of portability, security, and scaling, as they have to route their requests through their data store to get to other data stores.

Regardless of the logical solutions previously applied, the current systems do not properly address all of the aspects of creation, maintenance, security, and management properly. Because extensive time is required to master both an advanced data store system (such as a relational data store) and an advanced object oriented programming language, the costs associated with projects involving both systems are often viewed as expensive investments for the entities needing the development. Generally, object oriented application developers are not very intimate with advanced data store (i.e., relational) theory and performance tuning. Having a centrally managed and controlled relational access system is essential for organizations concerned with the efficiency and mechanisms for accessing data stores. There is no prior art system providing a much needed mechanism to completely separate the skills necessary to program with objects, from those necessary to create and maintain data store schemas and access code.

Accordingly, there is a need to dynamically build the code for accessing the data store at runtime, based on the object and its attributes, as opposed to embedding the access code within the object. There is also a need to reduce the maintenance costs of keeping both object and data access code up to date and in sync with one another in all of the application code, as well as allowing application data objects to readily work with any underlying data store regardless of any differences in their access methods and access code. There is also a need for such an approach in light of the explosive growth in the volume of new applications currently being developed for environments such as the Internet, which applications need to be dynamic and easy to manage. There is a need for a generic object oriented mechanism to bridge various underlying data stores consistently, especially in a transparent, generic, centrally manageable, and secure way. The present invention fills those needs.

SUMMARY OF THE INVENTION

The present invention comprises a system and method for accessing a data store having a data store content and a data

store schema as object(s) from an object application. The system comprises an object schema and an adapter abstraction layer. One embodiment of the adapter abstraction layer comprises a first adapter and a second adapter. The object schema includes meta data corresponding to the data store schema in some meaningful object-oriented view of data store content maintained at the data store.

One embodiment of the subject invention comprises the first adapter in communication with the object application and includes an application bridge receiving an object comprising object attributes and an object name. The first adapter also extracts the object attributes and the object name from the object to effect packing of the object attributes and the object name as data. Upon receiving new attributes from the data store, the first adapter also unpacks the data to effect instantiating the object attributes and the object name into a new object.

The second adapter is in communication with at least one of the first adapter(s) and in communication with at least one data store. The second adapter includes (or dynamically loads) a meta data map comprising object name(s) and provides the data store content from at least one data store corresponding to and based on the object attributes and the meta data.

One embodiment of the system uses an object schema manager to create and manage the object schema to be used at run time.

The present invention discloses an improved data access system and, in particular, a dynamic, scalable, centrally managed, and secure method for accessing data stored in both object and non-object (e.g. relational) data stores. The present invention effects a simple and consistent interface to at least one data store(s) regardless of its underlying structure, or method of transport and level of security implemented to access the data store.

BRIEF DESCRIPTION OF THE DRAWING

For the purpose of illustrating the invention, there is shown in the drawing a form which is presently preferred; it being understood, however, that this invention is not limited to the precise arrangements and instrumentalities shown. FIG. **1** illustrates an object application request as it is processed by the present invention, such processing in conjunction with a schema manager, and the way in which the present invention further processes the request.

DESCRIPTION OF THE PREFERRED EMBODIMENT

While the invention is described herein in connection with a preferred embodiment, it is understood that it is not intended to limit the invention to that embodiment. To the contrary, it is intended to cover all alternatives, modifications, and equivalents as may be included within the spirit and scope of the invention as defined by the appended claims.

An illustrative embodiment of a system and a method for accessing a data store as an object from an object application may be implemented using a product such as CocoBase Family of products, products of Thought, Inc., an assignee of the present invention, and the Java DataBase Connectivity Layer (JDBC), using a 3-tier architecture over a distributed network such as an Internet, Extranet, or Intranet.

FIG. **1** illustrates one such embodiment involving a request **100** by an object application **101** and the processing of such request. The system according to one embodiment of

the present invention comprises an object schema **200** including meta data **201** corresponding to a data store schema **300**, a first adapter **400**, and a second adapter **500**. In the preferred embodiment of the subject invention the object application **101** is a java applet "cocodemo.java" listed in appendix A.

One embodiment of the present invention uses a user defined object view of an underlying non-object store, i.e. the object schema **200**. The object schema **200** is accessed by an object application **101** through an abstract layer **600** which does the necessary conversions between object and non-object views. This abstraction allows both kinds of data stores—object and non-object (e.g. relational)—to be accessed identically from at least one object application **101**.

Moreover, by applying simple object streaming, the subject invention is capable of accessing objects distributed over a three tier environment using exactly the same application program interface ("API ") **700** as the local two-tier environment. This is accomplished by using access "adaptive" technology comprising the adapter abstraction layer **600**.

The adapter abstraction layer **600** of the subject invention performs any translation work necessary for converting objects to both object data stores **312** and non-object data stores **302**. The adapter abstraction layer **600** provides a consistent API **700** for both object and non-object data stores and enables application programmers to migrate between various object stores without application modification. The adapter abstraction layer **600** also facilitates communication with a remotely available second adapter **500** without modifying the object application **101** programming logic.

In one embodiment, the subject invention includes the use of meta data **201** (i.e. data used to describe other data) to define how to access and convert non-object data store content **304** to objects and back. This is accomplished by paring down the non-object data store schema **300** into its various components including tables, fields, and conditions **305** in one embodiment. The paring makes the creation, management, and access to the meta data **201** to be a convenient and elegant task. By using the adapter abstraction layer **600** which understands and uses the meta data **201**, the subject invention provides an abstract view of an underlying data store(s) **302** (**312**, **322**) as an object store. The effected abstraction produces an architecture whereby the underlying data store(s) **302** (**312**) as well as multi-tier adapters (e.g. **400**, **500**, **4**XX, and **5**XX) may be interchanged without object application **101** code modification.

One embodiment of the subject invention is implemented using the adapter technology and is therefore capable of being deployed as either a 2-tier or as a 3-tier environment transparently. The adapter abstraction layer **600**, and more specifically the first adapter **400**, communicates with a server process, more specifically the second adapter **500**, in communication with the underlying data store **302**, which transfers the requested data store content **304** to and from the first adapter **400** in communication with object application **101** client. The first adapter **400** instantiates the object **112** from more primitive datatypes comprising data **115**, reflecting the data store content **304** obtained from the server process, more specifically the second adapter **500**. The method of breaking down objects **102** (**112**) into the corresponding primitive types comprising data **105** (**115**) ensures successful transfers of any kind of object irrespective of object application **101** views of the object(s)' data elements. The subject invention uses the adapter technology to dynamically load data store access code **205** at runtime. This

architecture requires objects to serialize themselves in order to be persistable or for the computer language and language runtime to be able to inspect and set an object's structure in such a way as to be able to automatically persist the object's contents. Persistable is a term of art which means an object's contents can be saved to some more permanent location such as a computer disk, and in this case refers to the storage and retrieval of the object's contents from a data store. As a result, the present invention can readily adapt to an N-tier adapter solution which accesses the data store through some intermediate data server, thereby increasing deployment and scalability options.

FIG. **1** shows one embodiment of the present invention which works in conjunction with an object schema manager **203**. The object schema manager **203** allows data store administrators to create and maintain an object view of data store content **301** stored in the data store **302**. The object schema manager **203** reads, writes, and updates the meta data **201**. One embodiment of the present invention directed to accessing non-object (e.g. relational) data stores **302**, creates meta data **201** based on data store schema **300** comprising tables, fields and clauses or conditions **305**, making up the data store **302**. One such embodiment comprises a set of tools which create and maintain the meta data **201** by storing the information in a repository (such as the data store **302** itself), which information is then loaded by the second adapter at runtime. This meta data **201** then serves as a map for operations available on the data store(s) **302** (**312**), for the way these operations should be performed and under what conditions.

FIG. **1** further illustrates how the object schema manager **203** facilitates the creation and maintenance of the object schema **200**, thereby increasing productivity and reducing the opportunities for typographical errors, which can create preventable runtime errors. The object schema manager **203** permits dynamic modification of object schema **200** without requiring modification or recompile of the object application **101** to ensure the object application clients are not 'brittle' as is traditionally the case in data store access applications. One embodiment of the subject invention uses the meta data **201** to construct an object view of the data store schema **300**, e.g. an object schema **200**, and uses the meta data **201** to generate one or more commands **303** for accessing the data store content **304**. The deployment of meta data **201** in this way in one embodiment has the effect of using the non-object data store(s) **302** to be described and accessed as object(s) **102**, thought its contents may be more rudimentary and abstract.

One embodiment of the subject invention stores meta data **201** in a data store **204** and dynamically loads and executes meta data **201** at runtime. As a result, in one embodiment of the invention access code **205** corresponding to the command(s) **303** is centrally created and managed in a central repository, i.e. data store **204**. Such central repository architecture allows object schema(s) **200** to be re-used across objects **102** (**112**) and object applications **101** (**111**, **121**). Such approach also allows for any changes of one object schema **200** to be transparently reflected to any client object application **101** accessing and using the object schema **200**, without the need for object application **101** modification or recompilation. The subject invention contemplates a notification mechanism so that data store **302** modifications resulting from requests in one location be immediately communicated to all object applications **101** accessing that object schema **200**.

One embodiment of the subject invention facilitates the separation of data store access code **205** (such as Structured Query Language "SQL") from the object application **101** logic. This facilitates simple and rapid application development and lowers maintenance cost by improving the flexibility of the object application **101** and by reducing the brittleness of the object application **101** client(s). As a result, users of the subject invention attain a cost savings by centralizing the task, reducing the possibility of typographical errors, installation problems, and skills availability shortages. The object stores can be created and managed by someone with data store (i.e. relational) expertise (as most modern organizations have) without requiring those individuals to also have object programming expertise.

One embodiment of the subject invention results in an architecture for centrally storing and managing object schema **200** to promote re-use, effect enhanced performance, and better use of (MIS) engineering and administrative personnel skills. By way of example, storing the access code **205** within the physical confines of the non-object data store **302** in one embodiment, creates a centrally manageable and maintainable system for accessing that non-object store **302**. The object schema manager **203** thus allows for better use of the MIS personnel's skill sets by focusing on data access in terms familiar to people maintaining their respective system. That is, one embodiment of the subject invention allows object application engineers to concentrate on applications code without concern for or need to learn SQL or any other native access code **205** of the data store **302**. At the same time, data store administrators may concentrate on managing and maintaining the data store(s) **302** without knowledge of application programming and without concern that any changes in the data store content **301** or data store schema **300** will adversely affect the object application(s) **101** in communication with the data store(s) **302** according to one embodiment of the subject invention.

A central management architecture provided by one embodiment of the present invention allows data store specialists to see, manage and control all access to the data store(s) **302** down to the individual access code **205** commands. It also allows for the fine tuning and reconfiguration of those commands by the data store specialists, without involvement of object application developers. The architecture allows the best skills to complete the respective task at hand without impacting or being impacted by the lack of available skills that an individual may not have in the other area.

In one embodiment of the subject invention the object schema manager **203** allows for the arbitrary creation and management of object schema(s) **200** via a point and click interface, thereby significantly increasing the efficiency of creating the data store access code **205**, and improving the reliability of the overall system. This embodiment also allows completely arbitrary joining and combining of the data store content **301** as a single object view of that data, and overcomes the constrained and often uselessly arbitrary or non-existent foreign key constraints of conventional mapping systems and data stores. These hardcoded links such as foreign key constraints may not be supported properly by all underlying data stores, and not be implemented by the specific customer's data store (often legacy) implementation. New foreign key constraints are often not possible on some data stores because they may not work properly with older applications if the constraints were to be created and enforced.

As a point of reference, a foreign key is generally accepted as a predefined relationship or link which the data store manages in order to ensure that records are properly related in a database. A simple example would be to have a

foreign key between an order number in an order table, and an order number in a line item table which keeps the individual items belonging to that order. The foreign key in this example would be between the order numbers in both tables, which might not allow, for example, a line item record to be entered if there is no order in the order table corresponding to the order number.

A join using this embodiment may occur on any arbitrary field of any table within the data store schema **300**, even if a foreign key does not exist between the fields of the two tables to be joined. Such flexibility is immeasurably useful when combined with the ability of the object schema manager **203** to access more than one data store **302** at the same time. The resulting architecture of the subject embodiment allows data store content **301** from one data store **302** to be joined with data store content **311** from another data store **312** through the object schema manager **203** without requiring any data store schema **300** changes to either system whose content is being joined. Such transparent and independent flexibility is of utmost value for the many diverse data stores currently operated by most large and small business and other entities.

One embodiment of the subject invention provides a way to manage the data store access code **205**, requiring only an understanding of the data store **302** and the data store schema **300**, instead of requiring object programming knowledge within the modem working environment.

Once the object schema **200** is created, the object application **101** running in the application environment **103** sends a request **100** for an object **102** (possibly including an object data with lookup attributes to be matched) located in at least one data store such as an object data store **312** or a relational data store **302** in one embodiment of the subject invention. At a very high level, the request **100** is dynamically translated to a data store specific request which is then executed and results of which are parsed and new data object(s) **112** created and initialized with the results of the executed request **100**.

In one embodiment of the present invention, the request **100** and the accompanying object **102** is passed from the object application **101** to the adapter abstraction layer **600** comprising the first adapter **400**. The first adapter **400** then extracts the object attributes **103** and the object name **104** from the object **102**, and packs the object attributes **103** and the object name **104** as data **105** to be used in communication and transport layers. The first adapter **400** then communicates the data **105** and the request **100** to the second adapter **500**. The communication medium **630** between the first adapter **400** and the second adapter **500** may comprise an Internet connection, whereby the second adapter **500** is operating inside a firewall. Logical division **650** comprises a logical separation between a client, e.g. the object application **101**, and a server, e.g. the abstraction layer **600** and more specifically the second adapter **500**.

The second adapter **500** searches a meta data map **206** comprising at least one object name **114** using the object name **104** to determine whether the object name **104** exists in the meta data map **206**. If the meta data map **206** contains the object name **104**, then the second adapter **500** uses the object attributes **103** and the found meta data map **201** to generate at least one command **303** for accessing the data store **302** according to the request **100**, if the command has been enabled, ensuring further security by restricting access. By way of example, the subject invention contemplates the use of this technology to provide read only data stores over the Internet, irrespective of the user's standard privileges within the context of the data store.

The second adapter **500** then executes at least one such command **303** as a data store access client server process **500** in communication with the data store **302**, and obtains the data store content **301** and an execution status **306** based on executing at least one such command **303**.

The second adapter **500** then processes the data store content **304** and the execution status **306** using meta data **201**, and packs the obtained data store content **304** and the execution status **306** as data **115**. The second adapter **500** communicates the data **115** to the first adapter **400**.

The first adapter **400** unpacks the data **115** and instantiates the object attributes **113** and the object name **114** into potentially one or more new object(s) **112** and/or status according to the request **100** and the data **115**. The first adapter then communicates the request **100** and at least one such new object **112** and/or status, if object results were found, from the first adapter **300** to the application program **101**.

Several points should be noted regarding the implementation of the subject invention in the preferred embodiment. As each object(s) **102** data is being read from the data store(s) **302** (**312**), an instance of the object's class type (or potentially some generic container object) is instantiated (in the preferred embodiment a Java Class is instantiated for each set row result returned) and initialized to the row of attributes returned in the result set. Because Java can instantiate new objects based on classname, objects can be managed by the preferred embodiment of the subject invention even though it may have been unaware of their existence or composition at compile time. For ease of implementation, each java class in order to be supported by the preferred embodiment must implement a standard interface. With compliance to this standard interface, the subject invention can call the methods necessary to initialize the data automatically, making new instances automatic and transparent to the client object application **101**.

Implementation of explicit reference handling is not required using the subject invention, and therefore is not implemented in the preferred embodiment. Unlike more rigid systems which limit the type of operation and mechanisms which will and can be used to access data stores, the subject invention allows for flexibility. This is to reflect the market reality that most organizations operate large systems in place which should be capable of being accessed from a variety of sources in a variety of different ways. Any attempts to manage references in these mixed environments would not only prove a futile exercise, but would complicate and reduce reliability, without providing the desired functionality. As for reference handling within the object application, the Java language which was used to implement the preferred embodiment, has language supported reference counting within its memory management and garbage collection software. It is contemplated by the subject invention, however, that specific adapters and data objects may be implemented with this architecture for specific data stores supporting appropriate reference handling management.

The preferred embodiment of the subject invention uses an in-memory list to maintain object references, which list is either partially or fully prefetched. The object application may retrieve one object, all of the objects at once, or in blocks of default, or user defined size. Once the objects are in the memory list, they can be iterated through using their offset in the list as their index. The subject invention contemplates incorporating smart vectors which will automatically fetch Objects as they are requested from the vector itself, instead of the object application manually having to go back to the data store to request new blocks.

The preferred embodiment provides non-granular simple transaction support, although the subject invention contemplates either building it's own transaction management system, or using a pass through mechanism which allows for more sophisticated transaction management to be written by the application programmer, if the underlying system supports it. Any limitations due to the preferred embodiment being implemented in the Java Driver Database Layer and JDBC or the Java DataBase Connectivity Layer are not contemplated by the subject invention. Thus, the subject invention contemplates using more advanced transaction management, as available in the implementing language and architecture.

When the subject invention is used to access the non-object data store **302**, e.g. relational databases, the object schema **200** providing meta data **201** descriptions of how to map to the underlying data store **302** is used to generate the native access code **205** necessary to access the data store content **304**. One embodiment of the subject invention uses the object schema **200** to build an SQL statement at runtime, which is then executed to get the desired results, based on the request **100**. Because multiple statements are likely to be executed against a given class type once it has been accessed once, the present invention caches the meta data for the class (name) in memory, within the second adapter **500** and saves pre-compiled versions of the access code **205** which can simply be re-executed without recompile. This approach provides improved performance characteristics for successive accesses. In the three-tier design embodiment of the subject invention, the cache is physically located on the server, and is therefore shared across all client accesses to that server, providing a performance optimization in an area where performance is most critical.

The present invention comprises the adapter abstraction layer **600** comprising a set of runtime adapters (e.g. the first adapter **400**, the second adapter **500**, the n-th adapter **4**XX, **5**XX), which can be transparently interchanged by the object application **101**. The adapters serve various functions such as direct 2-tier data access, or 3-tier or even n-tier gateway/transport. Because all adapters implement the same API **700**, they are interchangeable in the object application **101**, providing new and useful functionality, even beyond the subject implementation, without requiring object application **101** modification. Object application **101** runtime flags can instantiate entirely new adapters which might interface with radically different data store(s) **302** without modification to object application **101**.

When data store access code, e.g. SQL, is dynamic verses pre-compiled, there is typically some performance penalties to be paid for dynamic execution. In the preferred embodiment, the subject invention addresses these performance issues by pre-compiling SQL the first time a given SQL operation is executed, and by preserving the compiled statement within the second adapter **500** for future executions from the same or a different first adapter **400**. The subject invention binds variables to the statement, and re-issues the generated access code with the new SQL. As a result, the penalty as to dynamic execution becomes a slight overhead the first time an object **102** of a given type is requested from the abstraction layer **600**. Past this first point the performance characteristics should be the same as from any other embedded SQL client, perhaps even better because the amount of information to be processed and transferred to the server accessing the data store **302** is likely to be less using the architecture of the subject invention than one which embeds SQL within a client object application and ships it with every request, and then must process the results inline, and often unnecessary descriptive data that is returned, and which is unnecessary in a system which is already preprogrammed with and understanding of the composition of the return values.

Because the first adapter **300** packs object request(s) **100** as a single API **700** call returning a single complex result (building the resulting data objects **112** transparently in the client from the returned complex data **115** packet), the subject invention reduces network latency (which is essential to reduce over the Internet and other io-bound and high latency networks) and increases the performance and scalability of the involved clients and servers. The subject invention also reduces the quantity of extraneous information which is always returned to client object applications by prior art systems such as the JDBC type-3 distributed database access layer architecture upon which other common Java Database Products are currently built. As a result, not only does the present invention obtain tremendous performance gains by reducing the latency of multiple network roundtrips, it also reduces the quantity of data and therefore increase the number of clients it can support, since each request/response requires less I/O. Compression and data duplication reduction techniques are contemplated by the subject invention to further increase, performance.

New adapters, e.g. **4**XX, **5**XX, may be added to the adapter abstraction layer **600** dynamically, even at runtime, as long as they comply with the API **700**. This allows data store(s) **302** (**312**) to have custom adapters built for them (such as object data stores or other relational and/or non-relational data sources) which comply with the API **700** of the subject invention and are loadable in the same way as the adapter abstraction layer **600** taught by the subject invention.

With the burgeoning of the Internet, secure access to data is essential. The subject invention implements security by using a novel approach in comparison to the current software trends and prior art systems. That is, instead of creating 'embedded' access code which goes into a client or a server at creation time, and which must be conveyed to the server, which therefore must be opened up to these general request, the subject invention teaches extracting, managing, and executing the access code exclusively and dynamically on the server for the described 3-tier system. As a result, one embodiment of the subject invention relating to relational data store allows an architecture which never exposes the data store's native SQL interfaces to the Internet. This approach eliminates the risk created by open SQL interface based clients. The only SQL which can be executed is controlled down to the table, field, and operations defined using the object schema manager **203** and is made available from only inside of the corporate network and firewall. This means that organizations can feel secure in knowing that hackers cannot steal passwords and create rogue, destructive, malicious, and/or prying clients which could potentially corrupt, steal, and/or destroy valuable corporate data, unlike other systems currently available which simply rely on passwords and encryption instead of also creating architectural obstructions.

As noted earlier, the preferred embodiment of the subject invention was implemented using the Java programming language, because of its dynamic language support, focus on security, and its overall suitability for the Internet. In the interest of fullest possible disclosure, the following appendices comprise the partial listing of the preferred embodiment.

CocoDemo is a demo applet which uses the CocoBase three-tier adapter API to connect. Appendix A contains

"CocoDemo.java" (Copyright © 1996, 1997 Thought, Inc., All rights reserved), the demo applet performing the object application functions and using the API, and "Customer-.java" comprising a data object which implements the CBDrop serialization interface, and can therefore be cracked open and constructed by the subject invention.

CocoBeanslmpl and CocoPowder together comprise the implementation of the second adapter and actually handle the server side communication and process the meta data and object attributes, and which executes the SQL. Appendix B contains "CocoDriverInterface.java" (Copyright © 1996, 1997 Thought, Inc., All rights reserved), which implements the API that the first adapter and CocoBeanslmpl/CocoPowder classes both implement.

The present invention may be embodied in other specific forms without departing from the spirit or essential attributes thereof and, accordingly, reference should be made to the appended claims, rather than to the foregoing specification, as indicating the scope of the invention.

8961-7
PBR/114837

-~~26~~- 20

APPENDIX A

```
/*

Copyright © 1996, 1997 Thought, Inc., All rights reserved

This class is a basic extension of the Applet class. It would generally be
used as the main class with a Java browser or the AppletViewer. But an
instance can be added to a subclass of Container. To use this applet with a
browser or the AppletViewer, create an html file with the following code:

    <HTML>
    <HEAD>
    <TITLE> A simple program </TITLE>
    </HEAD>
    <BODY>

    <APPLET CODE="CocoDemo.class" WIDTH=332

HEIGHT=169> </APPLET>

    </BODY>

    </HTML>

    You can add controls to CocoDemo with Cafe Studio.
    (Menus can be added only to subclasses of Frame.)
*/

import java.awt.*;
import java.util.*;
import java.applet.*;
import thought.CocoBase.*;
import Customer;

public class CocoDemo extends Applet {
    CocoDriverInterface myBase;
    public void init() {

        super.init();

        //{{INIT_CONTROLS
        setLayout(null);
        resize(374,262);
        c=new Label("Customer Demo Applet");
```

8961-7
PBR/114837

-27- 21

```
        add(c);
        c.reshape(56,8,224,22);
        label1=new Label("Name:");
        add(label1);
        label1.reshape(19,68,70,15);
        label2=new Label("Address:");
        add(label2);
        label2.reshape(21,98,70,15);
        label3=new Label("Phone:");
        add(label3);
        label3.reshape(21,135,70,15);
        edit1=new TextField(26);
        add(edit1);
        edit1.reshape(105,60,217,edit1.preferredSize().height);
        edit2=new TextField(26);
        add(edit2);
        edit2.reshape(105,98,217,edit2.preferredSize().height);
        edit3=new TextField(26);
        add(edit3);
        edit3.reshape(105,135,217,edit3.preferredSize().height);
        findButton=new Button("Find");
        add(findButton);
        findButton.reshape(14,202,56,30);
        insertButton=new Button("Insert");
        add(insertButton);
        insertButton.reshape(91,202,49,30);
        deleteButton=new Button("Delete");
        add(deleteButton);
        deleteButton.reshape(224,202,56,30);
        quitButton=new Button("Quit");
        add(quitButton);
        quitButton.reshape(294,203,49,30);
        updateButton=new Button("Update");
        add(updateButton);
        updateButton.reshape(161,204,49,29);
        //}}

        // Create an initial driver.
        myBase=CocoDriver.getCocoDriver("thought.CocoBase.CocoPowder",
                "jdbc.SimpleText.SimpleTextDriver",
                "jdbc:SimpleText", "myuser", "myuser");

        if(myBase.connect() == -1)
            System.out.println("Failed connect!");
    }
```

8961-7
PBR/114837


```
public boolean handleEvent(Event event) {
    if(event.id == Event.KEY_PRESS) {
        if(event.target == edit1 && event.key == '\t')
        {
            edit2.requestFocus();
            return true;
        }
        if(event.target == edit2 && event.key == '\t')
        {
            edit3.requestFocus();
            return true;
        }
        if(event.target == edit3 && event.key == '\t')
        {
            edit1.requestFocus();
            return true;
        }
    }
    if (event.id == Event.ACTION_EVENT && event.target == quitButton) {
        cleanup();
        return true;
    }
    else
    if (event.id == Event.ACTION_EVENT && event.target == findButton) {
        findCustomer();
        return true;
    }
    else
    if (event.id == Event.ACTION_EVENT && event.target == insertButton) {
        insertCustomer();
        return true;
    }
    else
    if (event.id == Event.ACTION_EVENT && event.target == updateButton) {
        updateCustomer();
        return true;
    }
    else
    if (event.id == Event.ACTION_EVENT && event.target == deleteButton) {
        deleteCustomer();
```

8961-7
PBR/114837

-29- 23

```
                return true;
            }
            return super.handleEvent(event);
        }

        public void insertCustomer() {
            customer = new Customer();
            customer.setName(edit1.getText());
            customer.setAddress(edit2.getText());
            customer.setPhone(edit3.getText());
            int insNum = myBase.insert(customer);
            edlg.displayMsg("Inserted "+insNum+" Object(s) into RDBMS");
        }
        public void findCustomer() {
            Customer keyCustomer = new Customer();
            keyCustomer.setName(edit1.getText());
            customer = ((Customer)(myBase.select(keyCustomer)));
            //customer = ((Customer)(myBase.call(keyCustomer)));
            if(customer != null)
            {
                edit1.setText(customer.getName());
                edit2.setText(customer.getAddress());
                edit3.setText(customer.getPhone());
            }
            else
                edlg.displayMsg("Customer  '"+keyCustomer.getName()+"'  not
Found!");
        }
        public void updateCustomer() {
            // New values for customer
            if(customer == null || (customer != null &&
                        !customer.getName().equals(edit1.getText()))) {
                    customer = new Customer();
                    customer.setName(edit1.getText());
            }
            Customer updateCustomer = (Customer)(customer.clone());
            updateCustomer.setAddress(edit2.getText());
            updateCustomer.setPhone(edit3.getText());
            int insNum = myBase.update(customer,updateCustomer);
            edlg.displayMsg("Updated "+insNum+" Object(s) in RDBMS");
        }
        public void deleteCustomer() {
            customer = new Customer();
            customer.setName(edit1.getText());
            int delRows = myBase.delete(customer);
```

8961-7
PBR/114837

- ~~30~~ 24 -

```
            edlg.displayMsg("Deleted "+delRows+" Object(s) from RDBMS");
            edit1.setText("");
            edit2.setText("");
            edit3.setText("");
        }
        public void cleanup() {
            myBase.close();
            System.exit(0);
        }
        //{{DECLARE_CONTROLS
        Label c;
        Label label1;
        Label label2;
        Label label3;
        TextField edit1;
        TextField edit2;
        TextField edit3;
        Button findButton;
        Button insertButton;
        Button deleteButton;
        Button quitButton;
        Button updateButton;
        //}}
        Customer customer;
        ErrorDialog edlg = new ErrorDialog("NONE");

    }
```

8901-7
PBR/114837

- 31 -

APPENDIX B

```
/*
 * %W% %E%
 *
 * Copyright (c) 1995-1997 THOUGHT Inc. All Rights Reserved.
 *
 *
 */

package thought.CocoBase;

import java.util.*;
import java.sql.*;
import thought.CocoBase.*;

/**
 * <H3>PROPRIETARY PROPERY of THOUGHT Inc.</H3>
 * <pre>
 * Copyright (c) 1996 THOUGHT Inc. All Rights Reserved.
 *
 * </pre>
 * CocoBase class (DataBase Catalog access class, designed for Catalog driven
 * access to relational databases, Meta Access Software System).<p>
 *
 * A class to access relational databases in a more object oriented fashion.
 *
 *
 * @author    Ward Mullins
 */
public interface CocoDriverInterface {
    /**
     * Post Constructor set methods for CocoBase.
     * Example usage:
     *          CocoDriver myBase = new
CocoDriver("thought.CocoBase.CocoPowder",
     *                          "jdbc.odbc.JdbcOdbcDriver",
     *                          "jdbc:odbc:Sademo", "sql", "dba");
     * @param myClass the Connection Class
     * @param connectDriver the JDBC Connection driver to be used.
     * @param connectURL the URL to use in connecting to the database
instance
     * @param userID the User Id to log into the database with.
     * @param passWord the Password to log into the database with.
     */
```

8901-7
PBR/114837


```
public void setDebug(boolean debug);
public void setUseBinding(boolean useBinding);
public void setUseSchemaPrefixes(boolean useSchemaPrefixes);
public boolean getDebug();
public boolean getUseBinding();
public boolean getUseSchemaPrefixes();
public void setConnectDriver(String connectDriver);
public void setConnectURL(String connectURL);
public void setUserId(String userId);
public void setPassWord(String passWord);
/**
 * Get a handle to the Connection object.  This will ONLY work in a
 * @return The Connection object.
 */
public Object getConnection();

/**
 * Do a commit on the database!
 */
public void commit();

/**
 * Do a rollback on the database!
 */
public void rollback();

/**
 * Set AutoCommit value
 * @param autoCommitValue The flag of whether or not to autoCommit
 */
public void setAutoCommit(boolean autoCommitValue);
/**
 * Return a ResultSet describing the columns which would be returned from
 * the Catalog manager for catalogName
 * @param catalogName the catalogName to find metadata for
 */
public Vector getColumns(String catalogName);

/**
 * Return a ResultSet describing the columns which would be returned from
 * the Catalog manager for anObject
 * @param anObject the Object to find metadata for
 */
public Vector getColumns(Object anObject);
```

```
/**
 * Connect to the database using the current connection information.
 */
public int connect();

/**
 * Fetch a Collection of all objects matching the criteria in myObject,
 * and using the definition associated with catalogName
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name).
 */
public Vector selectAll(Object myObject);

/**
 * Fetch a Collection of all objects matching the criteria in myObject,
 * and using the definition associated with catalogName
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name).
 */
public Vector selectAll(Object myObject, String catalogName);

/**
 * Fetch a single object matching the criteria in myObject, and using
 * the definition associated with catalogName.  If more than one object
 * matches the fetch, only the first is returned!
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name). this allows for multiple maps for the
 *  same Java 'object'.
 */
public Object select(Object myObject);
/**
 * Fetch a single object matching the criteria in myObject, and using
 * the definition associated with catalogName.  If more than one object
 * matches the fetch, only the first is returned!
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name). this allows for multiple maps for the
 *  same Java 'object'.
 */
public Object select(Object myObject, String catalogName);

/**
```

8961-7
PBR/114837


```
 * Fetch the Number of objects matching the criteria in myObject, and using
 * the definition associated with catalogName.  Return up to the number
 * of objects matched in by numberFinds
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name). this allows for multiple maps for the
 *  same Java 'object'.
 * @param numberFound the maximum number of objects to return.
 */
public Vector select(Object myObject, String catalogName, int
numberFound);

/**
 * Fetch a Collection of all objects matching the criteria in myObject,
 * and using the definition associated with catalogName
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name).
 */
public Vector callAll(Object myObject);

/**
 * Fetch a Collection of all objects matching the criteria in myObject,
 * and using the definition associated with catalogName
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name).
 */
public Vector callAll(Object myObject, String catalogName);

/**
 * Fetch a single object matching the criteria in myObject, and using
 * the definition associated with catalogName.  If more than one object
 * matches the fetch, only the first is returned!
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name). this allows for multiple maps for the
 *  same Java 'object'.
 */


public Object call(Object myObject);

/**
 * Fetch the Number of objects matching the criteria in myObject, and using
```

8961-7
PBR/114837


```
 * the definition associated with catalogName.  Return up to the number
 * of objects matched in by numberFinds
 * @param myObject the object to used as a 'template'
 * @param catalogName the definition name of object to be retrieved (if
 *  null use object name). this allows for multiple maps for the
 *  same Java 'object'.
 * @param procName the Name of the stored procedure to execute.
 * @param numberFound the maximum number of objects to return.
 */
public Vector call(Object myObject, String catalogName, int numberFound);

/**
 * Insert Object into Database
 * @param anObject the Object to insert into the catalog
 */
public int insert(Object anObject);

/**
 * Insert Object into Database Using Catalog
 * @param anObject the Object to insert into the catalog
 * @param catalogName catalog name of Object (if null, use object Name),
 * this allows for multiple maps for the same Java 'object'.
 */
public int insert(Object anObject, String catalogName);

/**
 * Delete Object into Catalog
 * @param anObject the Object to insert into the catalog
 */
public int delete(Object anObject);

/**
 * Delete Object into Catalog
 * @param anObject the Object to insert into the catalog
 * @param catalogName catalog name of Object (if null, use object Name),
 * this allows for multiple maps for the same Java 'object'.
 */


public int delete(Object anObject, String catalogName);

/**
```

8961-7
PBR/114837


```
 * Update Object in Catalog.
 * Both anObject and newObject should be of the same object type!
 * @param anObject the Original Object being updated
 * @param newObject the new value of anObject
 */

public int update(Object anObject, Object newObject);

/**
 * Update Object in Catalog.
 * Both anObject and newObject should be of the same object type!
 * @param anObject the Original Object being updated
 * @param newObject the new value of anObject
 * @param catalogName catalog name of Object (if null, use object Name),
 * this allows for multiple maps for the same Java 'object'.
 */

public int update(Object anObject, Object newObject, String catalogName);
/**
 * Close the connection to the Catalog, and to the Database.
 */
public int close();

}
```

I claim:

1. A system for accessing at least one data store having a data store content and a data store schema as at least one object from at least one object application comprising:

at least one object schema including meta data corresponding to the data store schema;

an adapter abstraction layer having a first interface responsive to the object application including an application bridge receiving an object comprising object attributes and an object name from the object application, said first interface extracting the object attributes and the object name from the object to effect packing of the object attributes and the object name as data, said first interface unpacking the data to effect instantiating the object attributes and the object name into a new object, and a second interface in communication with said first interface and in communication with at least one data store, said second interface having a meta data map comprising at least one object name and providing the data store content from at least one data store corresponding to the object attributes and the meta data.

2. A system according to claim 1 wherein the first interface comprises a first adapter and the second interface comprises a second adapter.

3. A system for accessing at least one data store having a data store content and a data store schema as at least one object from at least one object application comprising:

at least one object schema including meta data corresponding to the data store schema;

a first adapter responsive to the object application including an application bridge receiving an object comprising object attributes and an object name from the object application, said first adapter extracting the object attributes and the object name from the object to effect packing of the object attributes and the object name as data, said first adapter unpacking the data to effect instantiating the object attributes and the object name into a new object; and

a second adapter in communication with said first adapter and in communication with at least one data store, said second adapter having a meta data map comprising at least one object name and providing the data store content from at least one data store corresponding to the object attributes and the meta data.

4. A system according to either of claim 2 or 3 further comprising an access code used to access the data store wherein said access code is separate from the object application.

5. A system according to either of claim 1, 2 or 3 wherein said object schema comprises a possible arbitrary joining of the data store content across at least one data store.

6. A system according to either of claim 1, 2 or 3 further comprising an object schema manager in communication with at least one object schema and at least one data store schema, said object schema manager including constructs, facades, objects, and methods, to effect meta data based on at least one object schema and at least one data store schema.

7. A method for accessing at least one data store having a data store content and a data store schema as at least one object from at least one object application comprising the steps of:

creating at least one object schema including meta data corresponding to the data store schema;

communicating a request including an object comprising object attributes and an object name from the object application to an adapter abstraction layer having a first interface and a second interface;

extracting the object attributes and the object name from the object;

packing the object attributes and the object name as data;

communicating the data and the request from the first interface to the second interface;

searching a meta data map comprising at least one object name using the object name to determine whether the object name exists in the meta data map;

if the meta data map contains the object name, using the object attributes and the meta data to generate at least one command for accessing the data store according to the request;

executing at least one such command;

obtaining data store content and/or an execution status based on executing at least one such command;

processing the data store content using meta data;

packing data store content, if found, as data;

communicating the data and the execution status from the second interface to the first interface;

unpacking the data;

instantiating the object attributes and the object name into at least one new object and/or status according to the request and the data; and

communicating the request and at least one such new object and/or status from the adapter abstraction layer to the application program.

8. A method for accessing at least one data store having a data store content and a data store schema as at least one object from at least one object application comprising the steps of:

creating at least one object schema including meta data corresponding to the data store schema;

communicating a request including an object comprising object attributes and an object name from the object application to a first adapter;

extracting the object attributes and the object name from the object;

packing the object attributes and the object name as data;

communicating the data and the request from the first adapter to a second adapter;

searching a meta data map comprising at least one object name using the object name to determine whether the object name exists in the meta data map;

if the meta data map contains the object name, using the object attributes and the meta data to generate at least one command for accessing the data store according to the request;

executing at least one such command;

obtaining data store content and an execution status based on executing at least one such command;

processing the data store content using meta data;

packing data store content, if found, as data;

communicating the data and the execution status from the second adapter to the first adapter;

unpacking the data;

instantiating the object attributes and the object name into at least one new object and/or status according to the request and the data; and

communicating the request and at least one such new object from the first adapter to the application program.

* * * * *

EXHIBIT B

US006985912B2

(12) **United States Patent**
Mullins et al.

(10) **Patent No.: US 6,985,912 B2**
(45) **Date of Patent: Jan. 10, 2006**

(54) **DYNAMIC OBJECT-DRIVEN DATABASE MANIPULATION AND MAPPING SYSTEM HAVING A SIMPLE GLOBAL INTERFACE AND AN OPTIONAL MULTIPLE USER NEED ONLY CACHING SYSTEM WITH DISABLE AND NOTIFY FEATURES**

(75) Inventors: **Ward Mullins**, San Francisco, CA (US); **Alexandre Martins**, Centro (BR)

(73) Assignee: **Thought, Inc.**, San Francisco, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 293 days.

(21) Appl. No.: **10/624,753**

(22) Filed: **Jul. 22, 2003**

(65) **Prior Publication Data**

US 2004/0123048 A1 Jun. 24, 2004

Related U.S. Application Data

(60) Provisional application No. 60/397,837, filed on Jul. 22, 2002.

(51) **Int. Cl.**
H06F 17/30 (2006.01)

(52) **U.S. Cl.** ..... **707/103 R**; 707/10

(58) **Field of Classification Search** ..... 707/1, 707/3 R, 4, 5, 10, 100, 102, 103 R, 103 Y, 707/104.1; 345/736; 711/141; 709/201, 709/203, 225, 251; 713/201; 717/108, 114, 717/162, 165

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,754,670 B1 * 6/2004 Lindsay et al. 707/100

* cited by examiner

Primary Examiner—Apu M. Mofiz
(74) *Attorney, Agent, or Firm*—Robert G. Lev

(57) **ABSTRACT**

The present invention provides enhanced database access and performance when correlating or translating one database to another database or to an object programming application. The system and method of enhanced database access and performance of the invention provides a simplified high-level wrapper interface for global coordination of multiple software components and ease of use. Performance is also enhanced by utilizing an optional enhanced multiple user data caching system. The multiple user caching system of the invention provides a process of obtaining data from a data source, creating accessible data in the random access memory of at least one computer system the first time such data is read by a user wherein the data source corresponding to the cache is accessible to at least two users. Thus, the present invention is directed to dynamic mapping of databases to selected objects and a system and method for providing a multiple user caching system in such an environment or system.

16 Claims, No Drawings

DYNAMIC OBJECT-DRIVEN DATABASE MANIPULATION AND MAPPING SYSTEM HAVING A SIMPLE GLOBAL INTERFACE AND AN OPTIONAL MULTIPLE USER NEED ONLY CACHING SYSTEM WITH DISABLE AND NOTIFY FEATURES

This application is a provisional of Ser. No. 60/397,837 filed on Jul. 22, 2002.

TECHNICAL FIELD

The present invention relates in general to enhancing database access and performance when correlating or translating one database to another database or to an object programming application. The system and method of enhanced database access and performance relates to a simplified high-level wrapper interface for global coordination of multiple software components and ease of use. Performance can also be enhanced by utilizing an optional enhanced multiple user data caching system. The multiple user caching system relates to the process of obtaining data from a data source, creating accessible data in the random access memory of at least one computer system the first time such data is read by a user wherein the data source corresponding to the cache is accessible to at least two users. Thus, the present invention is directed to dynamic mapping of databases to selected objects and a system and method for providing a multiple user caching system in such an environment or system.

BACKGROUND ART

Computer databases or similar constructions (hereinafter referred to as data stores) are powerful tools for storage, organization, retrieval and other handling of various types of information. However, there are different database models, or formats, for data access that are incompatible with each other, and may also be incompatible with, or remarkably different from, an object programming application. In this respect, complex relationships between objects present in the object programming application may be entirely absent in a relational or object database being accessed or updated. Nonetheless, a plurality of these database types have achieved a high level of popularity and proliferation. The two most common database models are object and relational, and object programming models are frequently used for internet applications involving database accessing.

As an example of common object programming applications, the popular Java language with its server-side component, Enterprise Java Beans, is being widely used to support object model programming applications that access and use data from databases that have a JDBC driver. Thus, it is desirable to be able to adapt and use many of the traditional relational database tools, utilities, systems and corporate procedures with the newer object format of many more recent web-based applications. Since many users may be accessing a single application, there is a need for speeding up database accesses and for synchronizing object programming application transactions with database accesses and updates.

Significant drawbacks are associated with using a relational database model in conjunction with an object programming model, but the relational database model is still the most widely accepted traditional format for reliable and quick access to data while using complex data search queries. This model allows the user to store data records and

to access the data via a query language that uses relational expressions such as AND, OR, NOT, etc. Over the years, the vast majority of production databases have been relational databases, each having its own relational arrangement, and accessible by a particular query language. While convenient for set-up and arranging data, access is had only by those having substantial knowledge of the various relationships between pieces of data in the database.

JDBC drivers have been designed to formulate database specific SQL queries or statements from standard SQL strings passed to them by an object programming application. Ordinarily, such JDBC drivers are specific to a database, and separate statements and separate JDBC drivers must be utilized to access different databases. This is complicated by the fact that many object programming applications either do not permit simultaneous attachment to multiple databases or poorly coordinate transactions that impact multiple databases. Accordingly, relational databases can sometimes be awkward to access, and transferring data from one database to another database can be slow and painful, unless a specific transfer program has been specifically written or tailored to effect that specific transfer of data.

The object database model is a newer model that has rapidly increased in usage due to recent trends such as the world-wide-web and application service provider (ASP) architectures. Such object databases have sometimes been used as the primary source for accessing data, for example, Jasmine and ObjectStore are available from Computer Associates Intl., Inc. and Object Design Inc. (ODI), respectively. However, accessing data in such permanently stored object database has raised serious performance issues for larger applications when complex querying is required (as compared to relational databases). Complex querying is usually faster using relational databases since query engines may be more highly developed for relational database accesses.

Typically, a system that utilizes both a relational data source and an object programming environment involves coordinating, updating and synchronizing both a relational database structure and an object model or models that represent relationships between objects and data, objects and objects, objects and metadata, and the like. When data is being loaded, inserted, updated or deleted, object graphs must be navigated and multiple software components must be coordinated. There is a need for a simple, but high-level API and user interface that may be utilized sto coordinate multiple software components to synchronize and coordinate the relational data and object programming systems.

Further, regarding performance of such systems, object applications have previously tended to only use simple memory buffers with relational and object databases. Difficulties existed with using either an object database or a relational database to serve as a transient memory resident database "cache" (secondary database) to try to speed up database accesses. No commercially viable solutions have existed with acceptable performance.

Additional complications for the design of caching systems in an object language programming environment are caused by the need for multiple program processes or for multiple persons (both are generally referred to as "multiple users") to the same data source. If a single cache is used to place in random access memory of a computer system the complete data from a data source, then system memory overhead demands are high and expensive and system communication traffic requires constant update of the data source to keep it current for multiple users. When only a portion of a data source is cached as needed, multiple caches usually exist that correspond to each user and must be

synchronized (kept updated together) by updating everyone's cache (or everyone's cache containing the updated data) when data is changed. Such transaction coordination of caches and tracking the use of data in caches by multiple user creates large overhead which can slow or cripple performance on a larger system.

There is a need for an object programming application to successfully access a secondary transient memory caches of data, as raw data, relational data or object data for multiple users, particularly in an environment that may be distributed over multiple computer machines on the same system or distributed over multiple systems on an intranet or internet system. In such a case, a dynamic database mapping tool would be necessary to provide the object programming application with a map or maps to both the primary and secondary data caches in order to successfully implement dynamic transient memory resident caches for multiple users. It would be particularly helpful in optimizing the performance of an object programming application if the overhead could be reduced for a multiple user caching system while maintaining the reliability of the data in such caches, particularly where distributed caching is required for data retrieval and performance. In such a case it might be possible to directly access (and perhaps re-use) data objects stored in the multiple caches in a reliable but efficient transient resident memory caching system in an object programming language environment, including caching of the metadata for objects, instead of generating data objects from accessed data in a primary data source.

If it were possible to successfully provide a commercially viable secondary memory resident transient data caching system for multiple users, data being used by the object programming application would only need to be persisted back to one or more of a permanent storage relational database (primary database) when data has been created, inserted or updated. Ideally, the application would likewise update the memory resident caches with updated data only when the user needs to use updated data. However, as mentioned such simultaneous access of multiple databases or a single database by multiple users can give rise to transactional coordination problems, or to data synchronization problems.

Accordingly, intensely needed by many modern software application programs are systems for more efficient access and manipulation of data stores, systems having the flexibility and dynamic capability to attach data from a database to maps as objects and having the ability to map one or more databases to various objects in real time. A strong need exists for such systems that also permit a user to cleanly, transparently and synchronically transfer data between multiple data sources, while maintaining the ability for an object programming application to access or use such data in the system. The need exists for an improved database access buffering system, for object programming applications, having at least one transparent secondary memory resident database and a primary data source that can be simultaneously utilized in a synchronized and transaction coordinated manner.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the present invention to provide a simple wrapper or facade interface and API to be utilized to coordinate relational data systems and object data systems and to control multiple software modules that are utilized to load, insert, update and delete data from data systems and objects or object models in object programming systems during implementation of such functions in the overall system or network.

In a preferred object, the present invention provides a simple facade and object programming API designed to control the CocoBase O/R Software tools program components, particularly controlling the more complex components, to provide coordinated object graft and data persistence, including object model navigator components, transaction coordination components, CocoAdmin function components, and optionally caching components.

Another object of the present invention is to overcome the drawbacks of the conventional art and achieve flexibility and dynamic operation in the accessing of data by multiple users by providing an improved caching system for multiple users.

It is another object of the present invention to provide a system for dynamically mapping data to objects for software applications and to further provide a data or object caching system for multiple users having coordinated multiple caches, wherein the caching system may be in a distributed environment.

It is a further object of the present invention to provide a local or distributed data or object caching system wherein

(a) data or objects are preferably only placed in a user cache in the caching system when the data or object is first read from a data source by a user (lazy cache), or may be preloaded as directed by a user,

(b) each unit of data or object that is placed in a user cache is registered with a unit notification system (UNS) as being present in the cache and the UNS is connected to the Java Messaging Service (JMS) or to any other reliable messaging service available in the object programming language system environment,

(c) when any unit of data or object is updated in a user caches and persisted (saved) to a primary data source the UNS sends a message to all other user caches that may exist and contain the updated unit of data or object to register in the user cache a message that the unit was updated by another user and is noted as being invalid in the cache, and

(d) other user caches that have been notified as described in (c) will read from the data source the updated unit of data or object and update the cache only when a user attempts to access the invalidated unit or data or object in a user cache, may automatically update the user cache as soon as notified as has been pre-set in the caching system by the user of the user cache.

It is an additional object of the present invention to provide easy translations between databases and applications having a variety of different formats or data store models.

Another object of the present invention is to provide a mapping system wherein object programming applications are tailored to delegate accessing a database and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanisms in the application code. It is a preferred object of the invention to provide such a system with a concurrent parameter setting mechanism where the runtime library repository can be set to access a particular database and to generate database specific database calls and SQL strings.

In one preferred object of the invention, a software program would be provided that can generate application programming code from database maps to provide a programming application that will delegate to a runtime library

repository the functions of accessing a database and the generation of SQL strings that are specific to a database or JDBC driver, and coordinate such data and maps with one or more object models that can be created, updated, and navigated dynamically in response to changes to one or more of the data source, the mapping and runtime repositories, and one or more object models. In a preferred aspect, the software program has a coordination facility to persist any changes of data, objects or object models to a data storage facility and also provides a coordinated multiple user distributed caching facility.

In another preferred object of the invention, the runtime library repository can be modified and tailored to optimize database access calls and SQL strings for a particular database.

It still an additional object of the present invention to provide a mapping system wherein different maps for particular objects can be used to provide varying levels of security.

It is again another object of the present invention to provide a mapping system wherein data changes related to a particular object can be promulgated with global changes for that object, if desired. A particularly preferred object is to provide a multiple user caching system that may be termed a CANDON (Cache As Needed, Disable On Notification) caching system, where notification that a unit of data or object in a cache is invalid is provided to any user that has a cache containing that unit of data or object which has been updated by using a cache notification systems termed a UNS (Unit Notification System).

It is still a further object of the present invention to provide a mapping system when a data map for an object can be easily edited without extensive knowledge of the relational database as a source of the data.

It is again another object of the present invention to provide a mapping system wherein the metadata describing a map of a datastore can be dynamically evaluated.

It is again another object of the present invention to provide a mapping system, wherein data can be accessed more quickly than is possible with a conventional data store accessing arrangements.

It is again a further object of the present invention to provide a mapping system in which frequently-used data can be more easily accessed than other types of data.

It is yet a further object of the present invention to provide a mapping system in which a wide variety of different data languages can be easily used.

It is still a further object of the present invention to provide a mapping system wherein virtually any type of datastore architecture can be translated so as to be useful by an object software application, or other types of software applications.

It is again a further object of the present invention to provide a mapping system wherein datastore to datastore mapping is easily facilitated.

It is still another object of the present invention to provide a fully synchronized caching system that utilizes a messaging system such as the Java Messaging Service (JMS) to provide coordination facilities for a multi-user system or distributed caching environment.

In some objects of the present invention, concepts are based in part upon concepts present in U.S. Pat. No. 5,857,197, (incorporated herein by reference) or that are reasonably inferable from reviewing that patent, in order for the present invention to provide an improved mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about java objects including complex java objects that have relationships with other Java objects) does not need to be stored in a java object. In a preferred aspect, it is an object of the present invention to provide a database access system that does permits a java object application to delegate database accesses to a runtime library repository.

In a preferred object of the present invention, a software programming module (or modules) can automatically generate object source code from at least one database schema map, at least one object programming application schema, or from a combination of at least one database schema map and at least one object programming application schema. The code generated for the application can be set to delegate database access and SQL string generation to the runtime library repository instead of including within the application the more limiting code for non-delegated database access by the application. This arrangement allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. The mapping information can be used to map from objects to relational models or vice versa and generate appropriate code.

DEFINITIONS

For the purposes of the present application, the following definitions are given as a meaning for terms used herein throughout this application to avoid any confusion with possible multiple meanings for such terms. Other terms used herein have meanings that are well recognized in the art, and their meanings will be clear from the context in which they are used in this application.

A "module" in the computer programming context is an organized set of computer code designed to act on some externally passed in data, where everything needed for that action is passed in to the module.

An "object" in the object oriented programming context is an organized set of encapsulated programming code designed to act on itself at the request of some external system, which system may pass in some additional information to the object when it delegates a task to the object in that request.

A "composite object", or an object programming "component", in the object programming context each refer to an object comprising a complex and organized set of objects that are encapsulated to form the composite object. The two terms "composite object" and "component" (e.g., a J2EE component such as an EJB, Enterprise Java Bean) may be utilized in an interchangeable manner in referring to the same type of logical constructs and concepts.

A "delegation" in the object oriented programming context is where an object or a programming application permits another simple object, set of objects, composite object, or a module to perform an action by simply requesting the action from the delegated simple object, set of objects, composite object or a module.

A "non-delegation" database access in the object oriented programming context is where an object or a programming application has specific code imbedded in the application which directly controls database calls and the generation of SQL strings, wherein the imbedded code is specifically tailored for accessing a particular database or for a specific database schema.

A "user interface" for an object oriented application, such as a Java Server Page (JSP), a Swing GUI, and the like, refers to a software component or module that provides a feature for a user that will permit the user to interact with an object or programming application in some way, such as the interactions of finding, selecting, inserting, updating and deleting data in a database.

A "library" is a set of definitions, data, objects or programming modules that may be accessed by a computer programming application to obtain information or to delegate tasks.

A "repository" in the object programming context and in the context of this application is a special set of libraries that may include, among other things, items related to object to object mapping, object to relational mapping and database accessing information, database parameters, optimized database access routines, and the like. A repository may be a single file or may be a set of files. The format of items in a repository may vary widely according to the desire of a computer programmer user or developer and may be in one or more of formats such as simple text, XML, XMI, UML, JDBC, source code, compiled code, and the like.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is based in part on U.S. Pat. No. 5,857,197, (incorporated herein by reference), and provides a mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about java objects including complex java objects that have relationships with other java objects) does not need to be stored in a java object. Instead, the mapping information related to object definitions and some metadata can be placed in a separate structure that is independently stored. This allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. The mapping information can be used to map from objects to relational models or vice versa, objects to objects, object to COBAL or vice versa, and object to XML and the like.

Certain of the embodiments of the present invention are embodied in a suite of products by Thought, Inc., referred to as "CocoBase". Aspects of the CocoBase technology are also described in U.S. Pat. No. 5,857,197, supra. Also, a number of details of the system of the present invention are provided now as state of the art in the documents that have been posted on the website of Thought, Inc. or are otherwise available on the internet as publications.

In one embodiment, the mapping information, rules, or metadata can be maintained in a human-readable format and can be interpreted by an application to apply the rules to data objects. This information can be stored directly in the data source or can be stored in a separate file. In either case, this information is called a "repository". This repository provides an easily maintainable and transportable format for the mapping rules. The rules and java object relationships can be updated, or otherwise changed, dynamically even while other rules or object relationships in the repository are being used to perform conversions. The rules and relationship definitions can be embedded in standard formats such as in Java code, e-mailed, sent as plain text, etc. This allows for flexible transfer and proliferation of the rules to other sites where the rules can be used to access objects (which can be bundled with the rules), used to derive other rules, etc. In a particularly preferred embodiment, this repository is in an XML (extensible markup language) format, and the object model definitions can also be present in an XMI (XML metadata interchange) format or can be easily exported to an XMI file format.

A more preferred embodiment is such a system and software as provided by an updated CocoBase Enterprise Object/Relation Software package (hereafter CocoBase), which includes the software tool CocoAdmin. Such updated software package, is available, or is available shortly, from Thought, Inc., San Francisco, Calif. An embodiment of the invention using this package is described below.

Exporting Maps as XML Repositories in CocoBase

CocoAdmin provides a mechanism for the export of maps defined against a database into a modifiable XML format. This facility allows a system user to select an existing map(s) to be exported, and to also specify the filename of the XML document to be created. The resulting XML document is written to a file with the specified filename using an XML template named coco.dtd that is ordinarily located in the thought\cocodemo3tier31\demos\resources directory.

Once the XML file is written, it can be edited using a standard text editor or XML editor, and can be modified to reflect map customization requirements. The following discussion relates to specific features of the generated XML file.

The basic repository format has CBObject (CocoBase object class) definitions that reflect the select, insert, update, delete and call related map definitions for a CocoBase map. Each of those operations further consists of tables, fields and clauses that may exist to specify how the object is mapped to and from the data source.

XML files contain a DTD (document type definition) entry which describes the structure of the tags and data contained in the file. The DTD only checks for the existence of required elements, but doesn't check the contents of those elements. Given this desired degree of freedom, it is possible to enter invalid map information which will not be processed properly by the CocoBase runtime and care should be exercised to avoid improper editing. In general, modifications should be restricted to the schema, table, and field name variables in the Tables and Fields entries. These variables may require different values when they are exported from one database instance, and imported into another.

To begin exporting a set of maps, select File->Export XML Map Repository from the CocoAdmin pull-down menu. Multiple maps can be selected by holding down either the <Shift> or <Control> when selecting the connection and map(s) to export.

When Next> is pressed in the dialog box provided, the list of selected maps will be presented in a list and the window will prompt for a final acknowledgement of export. The XML repository filename including the directory path can be specified in this window. If the directory path is not specified, the XML repository will be written to the thought\cocodemos3tier31\demos directory. When the Export XML> button in the dialog is pressed, CocoBase creates the XML file in the specified directory. After the document has been written to the disk a dialog appears which acknowledges that the repository has been created.

Importing Maps from XML Repositories Using CocoBase

CocoAdmin provides a mechanism for importing XML based CocoBase map definitions, or for importing an XMI object model definitions and then generating corresponding

XML based CocoBase map definitions. Using the XML syntax defined in the thought\cocodemo3tier31\demos\resources\coco.dtd template file, CocoAdmin can import maps previously defined and exported from a different database or from a different object instance. The XML files generated from CocoAdmin will be validated against the DTD, and a basic syntax check will occur. No in depth syntax checking will occur because of the flexibility allowed in the system. When an import of an XML map definition occurs, it is in relation to an open database connection.

To begin importing an XML map definition from the CocoAdmin GUI select File->Import XML Map Repository from the pull-down menu or Import XML Map Repository from the popup menu in the main CocoAdmin dialog. Import an XML based map definition by selecting the database connection into which the import is to occur, and click the Browse button to find the XML repository file to be opened. After a file is selected the dialog button Load XML Document is clicked and the selected XML filename appears in the initial Import dialog. After the Next> button is pressed, the selected maps are compared with those already in the database. If a newer version of a map already exists in the database, by default a flag is set to retain the newer version. If the maps being imported are newer than the ones already in the database, then by default, a flag is set to import each of the newer ones. When the comparison operation is completed the user can override any flags before the task is initiated. For example, if the XML maps are older than the versions already in the database, this condition causes the Already Exists? flag to be checked and the XML map definitions will not be imported unless the user overrides this flag by checking the Import/Overwrite? box.

When a user clicks the Import XML Repository button as described above, maps marked for import will automatically be integrated into the list of maps in the CocoBase repository for that database. If the imported maps do not reflect the physical structure of the underlying database, they may need to be manually edited by the map editor before using them with an application. Also, XML format maps can be edited before they are imported. For example, an XML editor or any standard text editor can be used for this operation. Most common edits consist of changing column names, table names, and less frequently, the schema names of the tables and the fields that the map may access.

The CocoBase Programmer's Guide available at the www.thoughtinc.com website provides further information about how to work with XML format maps and CocoBase, which published document is incorporated herein by reference. This document also explains how to set CocoBase (or CocoAdmin of CocoBase) and its runtime modules to cache maps that an application will be using in order to speed up user access. Other user hints and instructions are provided therein. For example, one URL option in CocoBase is to specify an XML repository for the maps that an application will be using. Instead of using the database repository, the runtime can use an XML repository exclusively. This technique allows CocoBase to execute against a production database that cannot be modified with map definition tables. Optional mapping server plug-ins are also described.

Data source maps according to the present invention may be utilized to generate application programming code, such as Java source code. For example, the CocoAdmin tool will allow a user to generate Java for a range of targets including most commercial Application Servers and Enterprise Java Bean Servers. A complete list of pre-configured targets can be viewed from the Connections window in the Generate Java wizard of CocoAdmin. Any Java class can readily be used with CocoBase, whether or not it was generated by the CocoAdmin tool. However, there may be significant advantages that occur from using the code generation facilities of CocoBase to quickly generate your Java classes. For example, the disadvantages inherent with data source specific non-delegation database access code can be avoided by generating code, which delegates the database access functions to CocoBase runtime libraries. Only CocoBase is believed to utilize this type of code that results in dynamic O/R mapping system capabilities.

The CocoAdmin tool can use a combination of the map definition and database foreign key relationships (if any exist) to define how the Java class is going to be generated. The Generate Java wizard of CocoAdmin can be started in one of four ways to generate Java code from a CocoBase map.

1. Clicking the on the coffee pot icon located on the CocoAdmin tool bar.
2. Selecting File->Generate Java Code from the CocoAdmin pull down menu.
3. Selecting Generate Java from Existing Map from the launch pad after a connection has been made to a database repository, then clicking Launch.
4. Right clicking in the CocoAdmin main window and selecting Generate Java Code from Map from the pop-up menu.

From the connections directory tree in the Generate Java wizard, select the map from which a Java class will be generated. From this window, the CocoAdmin user can also set flags to automatically generate code that does one or more of the following:

Supports the transaction object (check TransObj).

Uses existing foreign keys to automatically determine object relationships (check ForeignKeys).

Uses the CBDrop, CBProp or both of CocoBase persistence interfaces (check CBProp and/or CBDrop).

The CocoAdmin user can also (optionally) enter in a package prefix at this point. Java classes are typically organized into packages, which are stored in a common directory. The package prefix (entered at the Pkg Prefix: prompt) will then refer to the directory in which the Java code will be stored.

The Code Generation Template drop-down list of the CocoAdmin code generation wizards allows a CocoAdmin user to select a target for the classes (Java code) that will be generated for a member of the comprehensive list of targets. The following target categories are supported for most commercially available application servers and EJB servers.

Java for distributed objects and application server applications.

Enterprise Java Beans with Bean Managed Persistence (BMP).

Enterprise Java Beans with Container Managed Persistence (CMP).

The Generic EJB Entity Bean CMP—All Parts option from the drop down of CocoAdmin will generate a completely standard CMP Bean, which can be installed using any vendor supplied CMP installer. CMP concepts for EJBs are discussed in a variety of documents published in this field, including in the appropriate section of the CocoBase Programmers Guide. In order to take full advantage of the CocoBase O/R mapping features, such as delegated database access and the like, a user must install the generic CMP with the CocoBase CMP installer tool, which is an optional component of the CocoBase software package. This tool will configure the CMP EJB to run in most commercially avail-

able EJB servers that support container managed persistence EJBs and coordinate the database access with the CocoBase runtime libraries.

Clicking Next> in the Generic EJB Entity Bean CMP—All Parts drop down wizards of CocoAdmin will present a user with a list of attributes which comprise the maps and any references (e.g. foreign keys) to other maps that may be navigated by CocoBase. For example, in a e-commerce shopping cart example, a Customer map, which is generated against a selected relational database connection, might be selected, and the PkgName: field might contain the name testpkg so the resulting Java code will be generated to a package and directory named testpkg. In a subsequent pop-up dialog, the user can add a foreign key reference by clicking on Insert Attribute and filling in the foreign key attribute in the inserted row. Such references can be added automatically for any operation which contains a join across tables of different maps. This topic is covered in more detail in the CocoBase Programmer's Guide.

Customizing Attributes to be Generated in Java Code by CocoAdmin

Attributes that are to be generated into the Java code by CocoAdmin can be customized through the code generation screen of CocoAdmin, which is displayed when a map is selected from the wizard connections tree and the Next> button is pressed.

For each fixed Map Attribute Label in the leftmost column of the table corresponding to a selected map, the corresponding Java Attribute Name and Field Type are editable. When the Field Type field item is placed in edit mode, the CocoAdmin user can select one of the available data types from the drop-down list. These relational data types will be mapped to a corresponding Java type when the code is generated. A relational to Java conversion table can be found in CocoBase Programmer's Guide, but any functional conversion table may be used. If foreign key references are present, each entry of the reference is also editable.

In general, the attributes for Java code to be generated should not be customized here unless a foreign key relationship needs to be modeled that isn't described by the database foreign keys or that isn't described correctly. Database foreign keys are automatically included in the map if the ForeignKeys checkbox in the wizard connections window is checked. Editing a map to include foreign key relationships is covered in the CocoBase Programmer's Guide.

A CocoAdmin user can specify a key for the map for the purposes of code generation by checking the box under the Key column. The attribute corresponding to the checked key box will be used as a key for the map and more than one box can be checked. For non-EJB applications, it is not necessary to specify a key, but keys can be useful in applications such as object caching and should contain unique values. Because EJBs require a unique key (for example, see the EJB spec. 1.0 from Sun Microsystems), CocoAdmin will not let a user generate code for an EJB without first specifying a key.

Customizing an Output Filename for Generated Java Code

When a map is generated into Java code by CocoAdmin, the output class name is specified manually by entering it in the File name field. If no filename is entered, the user will receive an error message when the user attempts to generate the Java code. Generally the user should adhere to Java naming conventions when naming package components. Package components should be saved in the PackageName directory and Java class files should have the form Classname or ClassName.java when entered into this field.

A CocoAdmin user can specify an existing output directory for the generated class (generated code) from the Look in: drop-down dialog. If from a previous window, the user specified a package prefix for the source code, then a subdirectory matching the prefix name can automatically be created in the specified output directory, if one does not already exist. Further information on naming and naming customization is presented in sections of the CocoBase Programmer's Guide.

Generating Java/EJB Code Using CocoBase

Once a filename has been entered as described above, a CocoAdmin user can generate the Java code by clicking the Generate button. When the Java Class or EJB has been generated, a dialog box indicating a successful generation will be displayed. If a user attempts to generate an EJB without specifying a key attribute, an Code Generation Exception Key must be specified! error message is received. Likewise an error message may be received if a file already exists with the filename that was specified by the user for code generation.

Prior to code generation maps of CocoBase may be customized to add relationships between fields, objects, and beans by creating links. Relationships between objects such as 1 to 1, 1 to many, and many to many may be set. Among other relationships, parent child class relationships are supported. Code may be generated which reflects such relationships. Also, during the code generation step relationships may be added using wizards and interfaces of CocoAdmin.

Link information can be dynamically entered and edited with a CocoAdmin dialog, or it can be loaded from the resources\CocoNavLink.properties file of CocoBase if code regeneration is going to occur often with the same class, and if keying in virtual foreign keys isn't practical. The database foreign keys are auto-detected at a table level, and can be edited using a CocoAdmin dialog if the Map name to be used is different from the default one generated using a table name.

Customizing Generated Code Through CocoBase Templates

The later versions of CocoBase Enterprise all have the ability to customize code generation through the use of multiple code generation templates. Templates are available from the tool through the CocoAdmin.properties configuration file. Each of the GENTEMPLATES name values found in a configuration file are comma delimited and specify the name of the code generation template to be used. The comma delimited file can be viewed using an ASCII text editor, for example. The "\" character which appears at the end of each line indicates a line continuation and is only relevant to the CocoAdmin tool. Name values present in this file can also describe a set of templates that will be processed simultaneously. The code generation list that appears in the tool by default includes, among other things, all of the major Java Application Server types, production of generic CMP, generate Java classes, generate JSP, and the like. See the current www.thoughtinc.com website for a more complete listing. The default Java object of the listing is a standard Java instance that implements the CocoBase interfaces. If a GemstoneJ EntityBean using Proxy objects is selected, for example, CocoAdmin will generate all of the GemstoneJ files necessary for a Bean Managed Persistent EJB Object. Such an "All Parts" template reference will actually cause several files to be generated. Such a selection would have a CocoAdmin.properties entry that specifies which template files are to be processed, and in what order to process them.

Such a file might be found in the thought\cocodemos3tier31\demos\resources directory of a particular version of the installed CocoBase software package. In one embodiment of the invention, when a particular template is picked any file prefix/suffix values to be appended to the filename would also be prefixed or appended to the map name automatically.

Mapping from One Data Source Format to Another Data Source Format

In another embodiment the invention provides a system for mapping from a first database format to a second database format, or from one database to another database of the same type, as a way for transferring data or synchronizing data sources. The system includes: data in the first database format stored in the system; rules for translating from the first format to the second format stored as a separate structure from the data; and means for applying the rules to the data to obtain the second format. This system may include a means for transferring the data from a first data source to a second data source. In one embodiment this system includes a simple computer program in a computer language such as Java, which reads the data from the first database using a repository map that may optionally be cached in the computer memory of the system and then stores the accessed data to a second database using the same or a different repository map.

An example of the structure for such a system and a short description/outline for a computer program adapted for such a system that is capable of transferring data between a first data source and a second data source might be described as follows:

First, let us assume two databases: a first database (B1) and a second database (B2) wherein B1 and B2 have the different corresponding data map schemas (S1) and (S2), respectively, and wherein S1 and S2 have the two corresponding schema repositories (R1 and R2) that include maps defining the structure (schema) of the database, Java object information (Java object model and any relationships between Java objects of the Java object model, or defining both the database and Java object information.

M1 is a map (or maps) defining the database schema S1 of R1 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B1, and

M2 is a map (or maps) defining the database schema S2 of R2 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B2.

The sample program outline,

(i) open a first system connection to access R1 and read M1

(ii) open a second system connection to access R2 and read M2

(iii) have a Java application create a Java object (Obj-1) in memory that can access the data of B1

(iv) allow the Java application to connect to B2 and access B2, or have the Java application create a Java object (Obj-2) in memory that can access the Obj-1 and also access B2,

(v) open B1 and B2

(vi) have Obj-1 retrieve data from B1 and place data in Obj-1

(vii) optionally have Obj-2 obtain data from Obj-1

(viii) have the Java application access B2 (optionally through Obj-2) and store the data of Obj-1 or Obj-2 in B2

(ix) clear Obj-1 and possibly Obj-2, or create a new instance of Obj-1 and possibly of Obj-2

(x) repeat steps (vi)-(viii) until all of the data is transferred from B1 to B2

(xi) close B1 and B2 and any other open files, and

(xii) end program.

In one preferred embodiment, step (vii) as described above involves having data retrieved by Obj-1 and then converted to an XML format data file which is then forwarded to a different location (and optionally simultaneously to a different user) on a distributed network, whereupon an existing or new instance of Obj-2 is populated with data from the forwarded XML format data file and Obj-2 then stores the data in B2. The XML format data file may be simply of a Java object, the data than can be stored in a Java object, or both the Java object and its data.

The above procedures implement a dynamic mapping layer in an exchange format such as XML that can directly exchange data from two data sources in a more dynamically controllable fashion to each other for the purposes of data import, export and exchange without the requirement of an intermediate third object. In such a model the developer would not be constrained by the structure of the database to which or from where the data is being transferred.

Also, the above example is an example of object to "DataSource" mapping. Data may be mapped to (or from) object and relational databases and may also be stored in XML. Also, XML stored maps can be utilized by CocoAdmin as a resource from which to generate object code for user applications. In one implementation, two system users (same or different system) using a single (or several major recognized) XML standard(s), can export or import (as XML files) either or both of the XML map definitions (relationships of data) and data (dataset) of the database. This would permit a user to distribute parts or all of a dataset (and/or maps) to a distributed network, or to multiple network users in useable XML format files. Such a phenomenon may be referred to generically as O/X mapping (object to XML) or R/X (relational to XML) in addition to O/R (object to relational mapping).

In one embodiment, O/X mapping step would permit importation or exportation of data (that was converted to XML) from XML to and from any business object. For example, data from any business implementation object could be exported as XML along with its relationships as defined by the mapping tool. This would implement a more flexible and dynamic mapping facility for taking XML datasets and using them to populate object instances other than the original ones that may have created the data. One good example of such an implementation would be to take a first XML document that is presented as a user screen interface to a dataset and utilize the information (dataset) populating the first XML document to seamless populate a second XML document (datasource to datasource via a XML (or another exchange format similar to XML) translation step.

There are many ways to utilize the above above implementations as part of either unidirectional or bidirectional mapping. Intranets and e-commerce applications can both take advantage of these important features. There are distinctive advantages when using a dynamic mapping layer where a java object provides translation by mapping such objects of a first data source (relational or object database) and also mapping such object to an XML or other second format data source. This allows more developer control as to how datasets are exchanged, filtered and/or validated between a first data source and a second data source.

One business use would be for a first party to use XML as a data export document to send information to a second

party (e.g., items offered for sale) and the second party would import that information in order to generate a second XML data document (e.g., purchase order) that the first party can import. This would provide a much needed exchange format for the newly developing business to business market that might be able to rely on standardized XML data formats and avoid problems caused by object programming models that may vary widely from company to company.

Such a dynamic mapping facility would provide a critical missing facility to easily translate data models and object model into a format that can be readily used by business to business companies, or by the same company for department to department interactions.

Documents describing the use of CocoBase products that do mapping of data sources other than relational databases have been published, and such products may be used in conjunction with the above embodiments. Examples of mapping include, mapping from object to object databases and from COBAL (mainframe computers of businesses often use this language data source) to object. See the IBM "white paper" on the topic of COBAL to object mapping (may be downloaded from URL http://www.thoughtinc.com/websphere.html).

In one embodiment of the system of the present invention a translation layer translates between an object application (or a potential object application, i.e. an object model) to at least one relational database which includes data entries organized as tables and records. Examples are database architectures supported by companies such as Oracle, Sybase, Informix, etc. Such an organization is well-suited for manipulation by relational query languages such as SQL. However, the traditional relational database organization is not ideally suited for manipulation by an object-based system. Some tools, such as Java Database Connectivity (JDBC) exist to achieve some degree of indirect relational to object mapping by accommodating some differences between an object model or application and the structure of the relational database. However, these tools also have drawback in that they are often not scalable, require extensive manual recoding for different object applications and are complex to use. Even tools (often mistakenly referred to as object to relational, i.e., O/R tools) that generate JDBC code (instead of generating a pure SQL string that a JDBC driver can utilize to subsequently create a pure SQL statement or query bundled in a JDBC format) are limited since they are often application specific, database specific, or specific to both.

In one preferred embodiment, a translation or abstract layer communicates with at least one JDBC (relational database driver) and at least one primitive Extended Java Bean (EJB) construct. The function of such a translation layer (generally called the O/R layer in CocoBase documentation, for example) is to translate object-based queries for the data into queries that JDBC can, translate into queries for a relational database. In a preferred embodiment, the translation layer can generate an SQL string (or strings) based upon the object-based queries, which can be passed to at least one JDBC, which JDBC can then generate an SQL statement from the SQL string. Similarly, the abstract layer accepts results from the queries and provides them to one or more of the EJB constructs in a suitable object format. The existence of JDBC is not necessary for all implementations of the invention. Also, different types of databases, data models, computation architectures, and the like can be used within the basic principle of the present invention. This mapping can be one-to-one, many-to-one, many-to-many, or any variation.

Another preferred embodiment of the present invention allows for mapping tables to be plain text, or to be text-based XML repository files. This not only allows the maps to be human readable and editable with standard editions, email programs, web browsers, etc., but allows for easy transportability and delivery of the maps.

Another feature of the maps is that they can be dynamically loaded into a computer system to provide for new dynamic mapping in a system that can that can continue to run while the new maps are loaded and executed. As described supra, the use of the translation maps provides advantages in organization, standardization of interfaces, efficiency of design, compatibility, scalability, portability and other advantages. This translation system of the present invention provides an entire set of tools to simplify and improve an operators ability to manipulate the maps without requiring a high level of programming knowledge or database query language familiarity.

The present invention provides a local or distributed data or object caching system wherein

(a) data or objects are preferably only placed in a user cache in the caching system when the data or object is first read from a data source by a user (lazy cache), or may be preloaded as directed by a user,

(b) each unit of data or object that is placed in a user cache is registered with a unit notification system (UNS) as being present in the cache and the UNS is connected to the Java Messaging Service (JMS) or to any other reliable messaging service available in the object programming language system environment,

(c) when any unit of data or object is updated in a user caches and persisted (saved) to a primary data source the UNS sends a message to all other user caches that may exist and contain the updated unit of data or object to register in the user cache a message that the unit was updated by another user and is noted as being invalid in the cache, and

(d) other user caches that have been notified as described in (c) will read from the data source the updated unit of data or object and update the cache only when a user attempts to access the invalidated unit or data or object in a user cache, may automatically update the user cache as soon as notified as has been pre-set in the caching system by the user of the user cache.

Such a caching system may utilize the JMS (Java Messaging Service) or other reliable messaging service available in the system environment as part of the UNS (unit notification system). The plug-in cache being used by the user records the notification that a unit of data or object/object model unit is invalid due to having been updated by another user. The cache can be set by the cache user to delete the invalid data, reload data from the primary data source or simply wait until the cache user needs to use the data before taking any action regarding the invalid data. If data is not needed again that is residing in a cache, notification of it being invalid with a setting of taking no action until the data is needed avoid unnecessary reads from the primary data source to update portions of a cache that may not be needed again by a cache user.

This CANDON caching system results in lower memory usage, less network traffic in a distributed or networked system and provides less chance of data corruption. Performance of the system can be enhanced significantly by utilizing such a system. Data source reads are reduced, memory overhead is reduced and network traffic is reduced. This is quite significant in a large size system.

Below is a general example explaining how a plug-in can be used as a database synchronization mechanism across multiple databases such as across a first database and a second database.

Example and Explanation for Using the CocoBase Plug-in Architecture to Synchronize Heterogeneous DataBases

Suppose the existence of 2 databases, B1 and B2, with different schemas, S1 and S2.

Assume both B1 and B2 have their own CocoBase repositories R1 and R2, where M1 is a map defined in R1 and M2 is a map defined in R2 and both M1 and M2 are optionally identified by the same map name.

Suppose that the system uses CocoBase (commercially available object to relational mapping tool having a mapping repository capability) and a connection of CocoBase with B1 is established as follows:

```
CocoDriverInterface connB1 =
CocoDriver.getCocoDriver(cbdriver,jdbcdriver,jdbcUrl +
";cocoplugin=thought.CocoBase.SynchPlugin",user, passwd);
"thought.CocoBase.SynchPlugin" is the name of the plug-in class that is
configured to replicate operations of database B1 to database B2. For
example, for the "insert" operation, the plugin implementation would be
similar to the following
(suppose replicationDB is set to connection to database B2):
    public int insert(int code, CocoUser userObj, Object keyObject,
String objectName, Integer objCount) {
        switch (code) {
        case CocoServerPlugin.CB_PRE:
            break;
        case CocoServerPlugin.CB_POST:
            Object o = null;
                if (objCount.intValue( ) >= 1) { // if object was
insert in B1
                    this.replicationDB.insert(keyObject,
objectName); // we insert it in B2
                }
            }
            break;
        default:
            throw new RuntimeException("Invalid plugin!");
        }
        return CocoServerPlugin.CB_CONTINUE;
    }
```

Once the above connections of CocoBase are established, any accesses to the database B1, such as:

connB1.insert(. . .)

will have the plug-in access B2 and replicate the operation that was conducted on database B1.

The same mechanism described above can be used to replicate the result of other database operations (e.g. update, select, delete, etc.) that are performed against B1. Further, the application that uses CocoBase maps to access B1 and B2 can have CocoBase set B1 as the primary database and B2 as the cache database. In such a situation only the cache database B2 needs to be accessed to retrieve data when data that the application needs is present in B2, but both data sources need to be accessed in order to insert or store data. This can significantly speed up data retrieve by avoiding disk accesses for data already present in a memory cache.

The snapshots or map modifications, or even general data of modification is facilitated through the use of caching. Since a variety of different types of data (including metadata maps, objects, production data, and the like) are used as options in the inventive system, a number of caching methods must be adopted in order to accommodate the various types of data. Examples of caches include: a write-through cache without transactional integration; a two phase commit integrated cache; and, a temporary working storage cache, which acts as a second database. All three of these techniques are discussed in more detail below. The present invention is also used to distribute caching or database using a plug-in Java database. Any number of different caching arrangements can be applied to the system of the present invention, but are limited only by what is possible in the art of data storage. The various caching arrangements can also be used to facilitate global changes in maps and data.

If caching arrangements are to be used with CocoBase, the accessing and updating of information must be defined in some clear way to control the caching and avoid unclear or unacceptable results. Discussed below are three methods for our caching controlling, followed by two examples of ways we cache in memory as either a "database" in memory with a third party plug-in or a "caching in memory of data" where both are connected to the O/R run-time layer. When a user connects to the O/R layer and subsequent database they register any optional plug-ins for caching or extending behavior and this then controls the access to data per one of the three mechanisms. Such options can be very powerful when combined with transaction capabilities of some Java application servers.

Write-Through Cache Without Transactional Integration

The first type of cache design is a "write-through cache without transactional integration" (no two-phase commit). This model assumes that applications not going through CocoBase may also be updating data in the first database (data source). In this method if a store data attempt fails at the first database, then the corresponding data of the second database (cached data) is deleted and may be (or may not be) refreshed from the first database depending upon developer policy. If the store data succeeds at the first database, then the corresponding data is updated in the second database (cached data). Notification of storage attempt results can be sent to the user to indicate a successful or failed attempt to store data. Therefore, multiple users can be utilizing the same cache in this instance and multiple applicationw (e.g, applications that do not go through CocoBase) may be accessing the first database and data integrity can be maintained.

Two-Phase Commit Cache

The second type of cache design is a two-phase commit cache design. This type of caching only allows updates of data by applications that work through CocoBase, through an application(s) that is either a compatibly integrated application(s) working concurrently with CocoBase, or uses the same policies of updating as CocoBase. In this cache model, data is only updated if CocoBase, or the integrated application can update the data in both the first database (source) and the second database (cached data). A commit to change data in both the first and second database (or information cache) is obtained as a first step and store data (update) completion is done in a second step. Again multiple users can use the same database source (or multiple application if they are integrated properly with CocoBase) and data integrity can be maintained.

Temporary Working Cache

The third type of cache design is a temporary working cache design. This type of caching only allows updates of data by applications that work through this temporary working storage cache (i.e., memory buffer or a second database which may be completely or partially memory resident database). This temporary working storage cache is designed to shield a first database from access and act as the

datasource to users. The updating and accessing of the first database can be defined by rules of the developer as they desire. This cache design assumes that database accesses by database information users are directed to the cached information through CocoBase or through another application. The information in this working storage cache is used to intermittently update or to not update the first database (data source) as defined by the developer. In such a model, transactional coordinated database updates and persistence should be synchronized by registering both databases with the application having transaction coordination, such as the J2EE server Weblogic 6.1, for example. In such a case, this cache design may be described as a synchronized transactional database caching model.

The CocoBase software suite provides distributed caching of a database by using a plug-in third party free java database (for example HYPERSONIC SQL), however any third party java accessible database could be used. For example, a CocoBase application user can define a map of a first relational or object database to be accessed, and then import the data into a memory cache through a plug-in java accessible database (second database). This can allows CocoBase or compatible integrated applications to control access to data as defined by a developer, and database users may re-directed to access the second database in memory instead of accessing the first database in order to speed up performance. This also permits a database to be distributed to multiple users or to multiple applications on a network, where CocoBase or pre-defined developer rules control the updating and persisting of data by storing back to the original database any changes made to the memory cached java accessible database according to one of the three caching methods described above.

A distributed cache may use optional mapping server plug-ins and control access to multiple server databases. In plain English, the data source object database such as Oracle is opened as a first server database, and a second server database (for example HYPERSONIC SQL) is also opened as a third party memory or virtual memory resident object database. The second server database can be loaded into a designated mapping server memory area. Once the first database is accessed, the data is sent to the second server database for storage and also sent to the user. Future accesses to that same data are directed by CocoBase to the second database. Also, updates of data are directed to both the first and second database by CocoBase and/or by a transactional coordinator of a J2EE server such as Weblogic 6.1 where both databases are registered with the J2EE server.

Alternatively, some of the latest alternatives for caching use the caching capability of a server to which CocoBase is providing maps and connectivity with a database, or the caching capability of another accessible caching facility. Any stand alone caching facility that a developer connects with the O/R layer as describe above in the caching methods descriptions can be implemented as a cache for database accesses).

A caching facility may be utilized by CocoBase to create a single or multiple objects in accessible caching facility memory, which object (or objects) may be stored in a relational database (or really any other piece of data, data object or data stream) object database, etc. for possible re-use or multiple use by application(s). CocoBase, integrated applications, or pre-defined developer rules will then control access to data and can permit the developer to require a user to directly access the data in cached memory rather than accessing the original data source. This can really speed up performance, e.g., up to 40 times faster with the combination of Oracle as the first database and Hypersonic as the second (cache) database than by using ordinary Oracle database accessing.

For optimizing performance of applications that utilize the CocoBase runtime libraries and other repositories, it is important to cache not only data, but also to cache the map (or maps) to be used in the CocoBase mapping runtime. The underlying mapping connection automatically does this in CocoBase. So for example if a user is accessing an instance of 'Customer', the underlying thought.CocoBase.CocoPowder software module will cache the map information for the customer map within the connection object.

In terms of CocoBase, map caching is when the software suite reads and caches the map created with a mapping administration tool such as CocoAdmin into RAM (random access memory) of the computer system. Once this map is cached, the CocoBase runtime repository libraries can generate and execute SQL much faster, but the first step of reading a map into RAM can be relatively expensive resource-wise. The cached maps are kept internally by CocoBase in a pre-compiled form that is exceedingly efficient for the runtime to access. Maps by default are cached within a connection once they are loaded and used for the first time. It is also possible to cache maps for all connections within a Java Virtual Machine (JVM) by using the CocoBase URL extensions for controlling cross connection caching.

There are two mechanisms that can be used currently with the CocoBase enterprise runtime libraries to control map caching.

The first is to use a cache configuration file that specifies which maps to pre-fetch and to place in shared cache. This type of map caching is controlled through a syntax such as:

JDBCURL:cocorep=configresource:/resourcefile.properties or for example

JDBCURL:cocorep=configfile:/opt/file.properties

The cocorep syntax can begin with a colon ':' or semicolon ';' syntax, so it can be compatible with different computer systems. The resource syntax describes what config resource should be loaded from the CLASSPATH, either directly from the file system or from some archived file such as a JAR or Zip file. The syntax of the Config file is that a list of MAPNAMES needs to be specified in a comma delimited form for the tag MAPNAME. A sample resourcefile.properties file might appear as follows:

```
# Map names to preload
MAPNAMES = Customer,com.mypkg.Address,com.altpkg.Order
```

Another type of CocoBase software suite map caching that can be done is based on an XML repository. A set of maps can be exported as an XML repository and either placed into an archived deployment file, such as a JAR or zip file, or the XML repository can be an un-archived file that is loaded directly from the file system. To specify an XML repository in the CocoBase URL, the syntax would look something like:

```
JDBCURL;cocorep=resource:/CocoRepository.xml
or
JDBCURL;cocorep=file:/mydirectory/CocoRepository.xml
```

Both the colon and semi-colon are supported as the separator character for CocoBase URL extensions. As a result ';cocorep . . . ' and ':cocorep . . . ' would be processed the same by the CocoBase runtime libraries. Some EJB descriptor syntax requirements place special meaning on the colon or semi-colon, so CocoBase makes it possible to use either of these two syntax choices.

Fortunately, map caching with CocoBase only needs to occur once a session or once a JVM depending on the option used, and then it is cached in RAM. This initial read and caching process should be taken into account if any performance tests are being created to test CocoBase. Using the per JVM map caching can be a great way to pre-load and pre-fetch maps so that performance numbers are accurate based on normal execution patterns.

The mapping server of CocoBase can be extended through custom developed classes called plug-ins. A CocoBase plug-in implements the Java interface thought.CocoBase.CocoServerPlugin. This is a standard Java interface, and any Java class, which implements it and provides a public default constructor can be loaded automatically by the server at startup. Examples of such plug-ins are the cache and connection pool that ship as source code in the demos directory of the CocoBase software tools suite.

A CocoBase mapping server plug-in will implement the following interface signature:

```
public interface CocoServerPlugin {
public int call(int code, CocoUser userObj, int objCount, Object keyObject, String objectName, Vector returnObjs);
public int close(int code, CocoUser userObj);
public int commit(int code, CocoUser userObj);
public int connect(int code, CocoUser userObj);
public int delete(int code, CocoUser userObj, Object keyObject, String objectName,Integer objCount);
public int getColumns(int code, CocoUser userObj, String catalogName, Vector colInfo);
public int getValue(int code, CocoUser userObj, Object key, Object value);
public int insert(int code, CocoUser userObj, Object keyObject, String objectName,Integer objCount);
public int rollback(int code, CocoUser userObj);
public int select(int code, CocoUser userObj, int selectType, int selectCount, Object keyObject, String objectName, Vector returnObjs);
public void setCommObject(Object comObject);
public int setValue(int code, CocoUser userObj, Object key, Object value);
public int update(int code, CocoUser userObj, Object keyObject, Object newObject, String objectName,Integer objCount);
}
```

In such an implementation, each method will be called before and after the associated operation (such as select) and the code will specify whether the action is a pre or post process operation also defined as public static codes in the plug-in interface API class.

```
public static int CB_PRE=0;
public static int CB_POST=1;
```

The Post condition will always be called unless the plug-in aborts or finishes the operation in the pre-condition of the plug-in. Otherwise the plug-in will finish its pre processing, continue into the actual mapping connection, and then call the plug-in again with the post processing flag. After the completion of the post-processing, the resulting Objects or return codes will be passed back to the client process.

If the plug-in wishes to intercept and 'finish' an operation in the pre-process state without actually having the Mapping layer become involved, then the CB_FINISH return code can be returned as the return code of the method. If there is an error in the pre-processing and it is desirable to abort the request once again without involving the actual mapping layer, then we can return a CB_ABORT code. Otherwise the CB_CONTINUE should be returned (an all is well code).

```
public static int CB_CONTINUE=0;
public static int CB_ABORT=1;
public static int CB_FINISH=2;
```

Since all select operations come through the single 'select' callback, a software application may need to know which method the client called. CocoBase uses a code to distinguish what the client actually requested the call with the following values. This tells CocoBase if a select, select-block, selectNext or selectAll was called by the client, and the software can respond accordingly.

```
public static int CB_SELECTONE=0;
public static int CB_SELECTBLOCK=1;
public static int CB_SELECTNEXT=2;
public static int CB_SELECTALL=3;
```

Since a callback may be shared information, CocoBase needs a way to determine the 'context' of which user and which connection the callback is dealing with. In each of the methods, an instance of the thought.CocoBase.Cocouser is passed into the plug-in. This is the context Object, which contains all of the information of the user, its underlying mapping connection and any other private data that the plug-in wishes to store in the Object. Each of the CocoUser instances has the following public 'get' interfaces (note that each has a corresponding set operation available):

```
public class CocoUser {
    public String getCocoConnection( );
    public String getCocoDriver( );
    public String getCocoURL( ) {
    public Object getDBConnection( ) {
    public String getpassword( ) {
    public String getUser( ) {
    public Object getUserInfo( ) {
}
```

The getUserInfo( ) method (and corresponding setuserinfo( ) method) is used by CocoBase to retrieve a plug-specific Object. This can be any object (include lists of other objects) that will prove useful in the plug-in. This provides a very simple and flexible system that is fully extensible, and still easy to program.

There are several examples of plug-ins included with the published CocoBase evaluation distribution, and analyzing the working programming source code included as part of the distribution is often the best teaching mechanism. The caching mechanism makes extensive use of many of the features described above.

Client Programming Applications Can Access Multiple Server Databases at Once

The mapping server provides a convenient platform to access and map and enterprise set of data sources, not just a single database. This provides a convenient system for porting data from one data source to another data source (see above).

Below are several examples of how a 3-tier client using CocoBase can access multiple databases from a single client object programming application:

1) The client can open up multiple connections to the same (or different) CocoBase Server—each having it's own database parameters and logins.
2) The server connection can access multiple physical databases through a software product called a 'Gateway'. These systems create a virtual relational database over top of a user's physical databases. Vendors such as Computer Associates, Oracle and B2Systems all have Gateway products that can be accessed from JDBC drivers, and such JDBC drivers can easily be connected to CocoBase.
3) If both sources being accessed are Oracle, Oracle DBLinks can be used to link in tables from other instances into the current instance being accessed.
4) Multiple physical connections can be accessed by using a plug-in which is easily obtained by extending one of the sample plug-ins shipped with the CocoBase evaluation distribution along the lines described above and in the instructions of the distribution, and loading the extended plug-in into the Mapping server.

A simple example for extending one of the shipped plug-in would be to have two open connections, one connection to a first database and a one connection to a second reference database. When the client wishes to first try to obtain data from the second database (for example, the second database is a faster or smaller memory resident database), a shared connection to the second database is left open in the plug-in, which would append a list of Objects available from that database to the getColumns lookup. The connection to the second database would integrate into the selected plug-in to provide a pre-process state so that data requested from the first database by the client program will cause the CocoBase runtime libraries to first look in the second reference database for an object that is present in the reference list of available Objects before attempting to access the first database that also has an open connection.

There are other ways for modifying the plug-in to accomplish the same result. For example, a plug-in could be modified to use the CocoUser storage area to store additional Connections or lists of connections to be accessed first in response to efforts by an application client to look up Objects.

Object Caching for Performance Enhancement

Object caching is provided by reading an Object (usually representing an underlying database row or fetched result set) instance and placing the data in memory for future reference while causing the database connection of CocoBase to remembers that the data is now in memory for future reference and it is unnecessary to access the database for that data. Using this technique, if the same Object instance is requested more than once by a programming application, it can be simply pulled from memory, instead of having to be retrieved from the physical database and the related overhead associated from that operation. Since subsequent reads are read from the in-memory copy instead of the underlying data storage system, this technique can be VERY efficient and highly enhance performance. If the database server is over burdened, this can become essential for optimizing and increasing overall system performance. Since the Object Cache allows users to retrieve data quickly and efficiently from memory, it can often avoid the need for object look-up again, even with multiple clients.

With CocoBase Object Caching, as the data for each object (or the object in the case of accessing an object database) is loaded from the data source, the loaded object is stored in an in memory object cache. The Object must be uniquely identifiable to be cached—in other words a 'unique' primary key field or composite key fields must be defined. This cache is shared by all of the clients accessing the Mapping Server, and, as a result, only one copy needs to be in memory at once.

The CocoBase Object Cache contains a method for looking up the object in terms of its unique object identifier column(s). Before an object is loaded from the database by the CocoBase runtime repository, it is looked up in the cache by the identifier column(s). Using this technique, an object present in the cache is loaded from the cache instead of requiring the system to access the data source and create a new instance of the object. If the object is found in the cache, then that object is returned from the cache. A developer could create the cache in a way that would cause it to grow and fill all available memory, so it is wise to use a large memory size for the Java startup parameters. If a VERY large set of tables needs to be cached, then something like the Gemstone/J Java virtual machine should be used. Their virtual machine can grow with its own virtual memory, allowing the cache to grow as needed.

Caching with the CocoBase Enterprise Optional Mapping Server

CocoBase Enterprise provides two separate examples of a shared central cache source code examples of optional mapping server plug-ins. These programs are CocoServerSamplecache.java and cocoserverSamplecachePool.java, and each is included as both source and pre-compile binary forms. The plug-ins can be modified for custom caching, or they can used as they are shipped to provide useful out-of-the-box shared caching. The shipped compiled binary code classes belong to the thought.CocoBase package by default, and should be copied into the subdirectory ..\classes\thought\CocoBase if the developer finds it necessary to modify the source code and then re-compile the modified code to provide new pre-compiled binary forms.

In the demos directory of the CocoBase package there is provided a cache configuration file, which can be used to configure Map keys that will be used by the system for caching features. The file CocoKeyBundle.properties found in the demos directory of the installed CocoBase software package, can be edited to customize caching easily. Just add entries for each Map which describe the key or keys of that map, which will comprise a unique 'key' utilized in identifying each unique instance. The format is 'Map=COL[,COL2[,.]..]'. To provide an Index for the Map 'Customer' using both attributes 'name' and 'address' as the composite unique key, place an entry in the CocoKeyBundle.properties file which has the following format:

Customer=NAME,ADDRESS

Note in the above entry that the attributes are ALWAYS upper case, and that they match the 'virtual' column name used in the map (for example :NAME and :ADDRESS even if columns are NAM and ADDR).

The client (presuming they have permission through proper assignment of their admin ids by system administra-

tors) can "additional override" this default value, but using a CocoKeyBundle.properties file will allow the caching to be configured on the server without having to code or make client specific changes. By using this default properties file, each column that is included in the properties file is made part of the lookup key by the cache.

In some cases, it may be desirable to allow the client to configure a cache, and the following example code accomplishes the same task. This example code is extracted from the demo program CocoDemoCache.java included in the demos directory of the installed CocoBase software package. The example code uses the CBSETCACHEKEY "virtual Object" and "procedure call", both of which can be intercepted in the caching plug-in, and the plug-in can be set to over-ride each of them to manage caching configuration.

```
//
// Set up a searchable cache key for the Customer Map.
// Since the demo database doesn't have primary keys supported
// in JDBC, we can simulate a primary key by configuring.
//
Vector cbkeyfields = myBase.getColumns("CBSETCACHEKEY");
  if(cbkeyfields != null) {
    System.out.println("Server is cache Configurable, configuring
Customer object cache.");
      // Configure Key information for Customer so server caches it
      // properly. We only have to do this because
      // SimpleText doesn't support primary Key fields. If it
      // did, this wouldn't be
      // necessary.
      GenPObject gpo = new GenPObject( );
      Properties props = new Properties( );
      // We want to index on the NAME attribute
      props.put("KEYLIST","NAME,ADDRESS");
      props.put("OBJNAME","Customer");
      props.put("CBADMINUSER","cbadmin");
      props.put("CBADMINPASS","cbadmin");
      gpo.setPropObjectData(props);
      myBase.call(gpo,"CBSETCACHEKEY",0);
  }
  else
    System.out.println("Server either doesn't support Caching or isn't
cache configurable.");
```

While it is much simpler to configure caching on the server through the properties file (and preferable) as described above, client side configuration is also possible using the technique described above. Both plug-ins keep the cached objects in a local in memory server cache until the changes have been committed into the database. Once the changes to objects are committed into the database, the cached objects are merged into the central shared cache and are then accessible by other clients needing those objects.

Performance can be enhanced by maintaining a list of previously instantiated objects in an "in memory" cache list of objects in the Optional Mapping Server. This can be readily accomplished through using a variety of data structures and facilities including Hashtables, Vectors and Object databases such as Gemstone (which use a technique called "persistence by reachability") to extend the Java Virtual machine to have Virtual Memory.

System Performance Increases are Possible with Caching and CocoBase

CocoBase can support several thousand direct database requests per minute even on a simple Pentium server with garden-variety databases. Caching provides an even better scaling promise, especially for heavily accessed databases. Services such as Caching and connection pool management can significantly enhance performance numbers. Tests of the caching methods when CocoBase was being improved to implement the above concepts have shown that performance ca nearly triple through the use of even simple caching techniques. For example, some internal benchmarks with CocoBase and caching have clocked a 233 MHz K6/Pentium processor machine with Windows NT and 64 MB RAM as being able to serve up about 20,000 database requests per minute for individually requested cached data objects. The 20,000 requests did not constitute high volumes of data (about 30 bytes per request—a simple Object), but nonetheless such benchmarks can give developers a "rule of thumb" idea of potential increases in throughput due to caching, even when a very low-end and non-tuned system is used to serve up responses to database requests.

Other tests with CocoBase and caching used a small matrix test suite to show approximately how many cached read operations per minute a developer might be able to expect on a sample Platform. The numbers also indicated how accessing blocks of data can actually increase the overall network throughput of objects that a server is capable of handling. For example, blocking data in access blocks of 100 instead of access blocks of one can actually increase the server throughput to 5 times the data volume even when though the number of actual requests, which could be served, was reduced. These tests might encourage developers to use the block API wherever possible, and then tuning the access block sizes to an appropriate size for an individual application to enhance performance.

In some caching tests, a block size of 20 had a higher throughput overall than a block size of 100, a fact that may be related to the mechanics of how CORBA handles large blocks of I/O.

The above performance numbers might be merely considered guidelines, since performance numbers such as this can be very network, ORB and data specific. However, these tests do clearly demonstrate that using CocoBase with caching (Mapping Server Cache) in combination with the tuning of block size can have an significant impact on performance, and can sometimes provide a significant throughput increase for application/database servers.

There are obviously additional ways to tune specific environments and configurations even outside of CocoBase, since the above numbers are mainly given as approximations that demonstrate the CocoBase system and technology that can give developers real world examples of tuning performance, and perhaps a ballpark idea of what might be possible. Performance enhancement of an individual system and application configuration may vary based on Java Runtime, OS, Database(s) used, networking configuration including and bandwidth capacity, ORB, etc.

Where caching really can really be beneficial to a system is in the client's 'perceived' performance, and when 'blocks' of data can be successfully cached (i.e., the application does not need to wait for an SQL statement to execute in order to access and use data for processing a request, and the system does not have to wait for the same unchanged records to be accumulated in memory over and over again), a system can seem faster because of the reduced processing delays. The CocoBase system uses caches that are written to serve multiple users out of a single pool, and by default the CocoBase cache system is a write-through cache for extremely scalable performance numbers.

Such a write through cache is written for a CocoBase system designed to provide scalable performance while avoiding certain undesirable impacts on the transactional capabilities of individual application servers. Since caching is a complex science, it should not be tackled at the beginning of a project that uses the CocoBase system.

Instead caching should be implemented and fine-tuned as a final step in development. Such techniques can allow a developer to refine and improve an already streamlined system and avoids using a cache during development that could lead to unfortunate masking of poor software designs and system design flaws. Such problems might only become evident when the final application is implemented and scaled, if performance issues arise, which were masked by the caching during development stages.

Enabling Optional Mapping Server Object Caching.

In the Optional Mapping Server that can be used for object caching, caching is implemented in the Server Plug-ins instead of being within the Optional Mapping Server itself. This can provide more flexible caching that is user extensible. If either the CocoServerSampleCache or CocoServerSampleCachePool plug-in is loaded, all objects having a database primary key that is accessible from a JDBC will be cached by the CocoBase system. In this system, object data caching is done through the Optional Mapping Server plug-in APIs. Source code jump-starts for providing caching are included with the Optional Mapping Server as source code, but the CocoBase system does not automatically integrate caching directly into the Optional Mapping Server. Such integrations should be saved until the later stages of project development to avoid masking design flaws.

At the final stages of development, a developer can utilize the source code provided with the CocoBase package and implement a specific kind of caching necessary for the design of the developed system, and integrate that caching system into the Mapping Server. As described above, when the source code of the examples provided don't provide explicit functionality desired by a developer, the code is extensible and can be modified to meet specific project requirements.

Sample cache provided can yield many of the common features needed by developers when the CocoBase system is implemented, and the sample cache will probably be completely useful without modification for many tasks. Modification and implementation of caches are an advanced developer functionality issue. The steps of implementing and tuning caching should only be taken on as a final step before deployment, and be taken on with careful planning. Blindly implementing caching without fully understanding a system design and understanding the purposes for caching in the context of that system design can create serious performance, memory and data integrity issues. For example, having multiple databases open in combination with caching can raise real issues with respect to data integrity if caching and multiple databases are not managed properly, particularly where database accesses are not properly managed and synchronized.

Managing Object Integrity for Applications Using a CocoBase System

Object integrity is typically maintained by keeping the cache synchronized with the data in the underlying data store, and by ensuring that any changes are reflected in the cache if, and ONLY if, such changes are successfully committed to the database. When a programming application ends a transaction with a commit operation, any updated persistent objects are written back to the data store, locks are released, and the client specific cache is invalidated (this is referred to as integrating the client specific cache into the shared cache). When the application ends a transaction with a rollback operation, updates are discarded, locks are released, and the client specific cache is invalidated. This will leave the shared cache unaffected and unmodified.

For some application environments, objects may additionally need to be retained in the cache across transaction boundaries (in other words it may be desirable to integrate a change into the shared cache before it is committed). In this case, some application-specific logic is needed to deal with the need for consistency between the objects and the underlying data store.

Another important issue associated with Object Integrity is the need for unique Object Instances. If duplicates are allowed, it can become nearly impossible for a system to cache and to track Object Instances without the use of the transactional capabilities of a server to synchronize the cache and the main datasource. Because of this, some 'composite' key may be required to ensure that unique instances exist. If a unique key definition exists, then the Mapping Server can manage Object Integrity more properly.

Differences Between Map Caching and Data Object Caching.

Object Caching and Map Caching should not be confused. Object Caching caches the user data, and must be implemented in a Optional Mapping Server plug-in. Map Caching automatically occurs in a CocoBase connection. Since Maps can become quite complex depending on the number of where conditions and fields involved in its definition, it becomes critical to Cache connections by default. As a default, CocoBase meta data for each Map is cached by the CocoBase system the first time that the Object is accessed.

The meta data caching can be triggered for a particular map before the data from the map is utilized by calling the method getColumns( ) on the Connection with the Map name as the parameter such as:

Vector tempvect=myBase.getColumns ("Customer");

In this example, a Customer Map will be accessed and cached in the CocoBase runtime before any SQL is created or allowed to execute. This is a convenient method designed to speed runtime access of Maps in the Mapping Server through pre-loading and caching the Map. If this pre-loading and caching is necessary for Maps that are likely to be accessed by clients, it can be done at server/connection startup in order to improve perceived user performance characteristics. Once the server/connection is achieved the system is ready for database accesses, there is nos second delay required to pre-load and to cache the maps before accessing data.

Because some systems may have literally thousands of Maps for a single connection, the CocoBase runtime is not configured by default to auto-load all maps, only those specifically requested to be loaded or accessed directly will be automatically loaded. In the case of Map caching, a lazy cache is more effective. Especially since with a lazy cache it is so easy to trigger a load and cache of particular Maps that are likely to be used.

CocoBase is currently configured to only permit an application to dump a Map cache from a connection by closing and re-open a connection. Such a dump involves only two method calls, and is easy to implement if necessary by simply entering calls similar to the calls to the "MyBase" database that following two lines illustrate:

```
MyBase.close( );
MyBase.connect( );
```

These two lines will cause all of the Meta Data Map information related to this database to be reset from the connection.

Object Uniqueness Requirements in a Cache

Since it is possible with most databases to have duplicate records in a table that maps the data structure, this can prove problematic for providing a caching function that can server up unique Object Instances. The CocoBase system and runtime resolves this issue by using a default setting with respect 'unique' keys. This default setting will only permit caching of those records that are specified by a 'unique' key, which identified through the Database Meta Data available from JDBC. Additionally, it is possible for a developer to define a unique key column (or set of columns) through a startup init file or through the runtime API. Only a developer with advanced system design understanding and high level programming skills should attempt to make such a modification, since this is a task that should only be done if instances to be stored in the cache through the map are truly unique.

The CocoBase system is designed to manage the cache by using a unique key, or the key column(s), as the unique identifier in looking up instances and then stores the instances in a Java Properties Object with the unique key as the identifier. With this design, if two duplicate records are retrieved from the database, only one instance will be preserved in the Properties based cache. The CocoBase is not designed to create uniqueness or to enforce uniqueness for database tables that do not have unique index defined. Because of this, the CocoBase system is designed to ignore an attempt to create caching or to use caching if a unique primary key has not been defined in the database, or a unique key is over-ridden by the developer by their software application design.

The CocoBase system is designed to allow non-unique objects to be accessed just as in ordinary database accesses where the CocoBase system is not used. However, the use of as caching is not turned on for those Objects. If a database table doesn't define a unique key, the Objects simply get passed through the CocoBase plug-in without caching becoming involved. However, a developer can easily take advantage of CocoBase plug-in caching even if a database table does not use a 'unique' key column set, by simply defining their own unique key for the database table.

Developers can look at the CocoKeyBundle.properties startup init file to see how the CocoBase system provides an ability to dynamically define a Unique Key. Also, developers can review the CocoDemoCache.java file portions related to the CocoBase connect( ) methods format and commands, for an example of how a developer can easily define an Object Unique key through the use of the Mapping server CBSETCACHEKEY functionality.

Design of the CocoBase Sample Optional Mapping Server Object Caching

As described above in several text locations, there are two sample Caching plug-ins included in the CocoBase system distribution and the design of each plug-in is bid different. One plug-in is a "Cache only" plug-in with the file name CocoServerSampleCache.java, but the other plug-in named CocoServerSampleCache.java is designed to provide a combined caching and connection pooling.

The cache portion for each plug-in is basically identical to one another, and the cache design is fairly intelligent with respect to caching systems. The cache portion of the plug-in keeps a rollback log for each client that is using the cache (a private cache) which is integrated with the shared cache upon the completion of a successful commit that updates the datasource. If the cache setting of AutoCommit is used, the caching system of the plug-in is designed to automatically integrate the changes to the private cache into the shared cache after each operation is completed that changes the private cache.

The CocoServerSamplecache plug-in is loaded as the default plug-in for the script startcache.{bat/sh}. The other plug-in is used by substituting the script startcachepool.{bat/sh} and CocoServerSampleCachePool is loaded as the CocoBase system plug-in instead of the default cache plug-in.

The caching portion of the plug-in is also designed to support some administrator control functions for starting, stop, and managing the cache. Additional features may be added by modifying the source code that is provided as part of the CocoBase system. The cache plug-in source code described herein (and to be released after the filing of this document) supports (among other things) the administrator control functions of CBSTARTCACHE, CBSTOPCACHE and CBSETCACHEKEY, which can be used by a developer to set a caching system design that will start and stop the shared cache, as well providing the ability to override the primary key column(s) used at a resource to lookup Objects in the cache.

The cache plug-in source code does not need to be modified to manage transactional coordination or regulate cache memory size management. Cache system design is easily configured to solve such issues in a much more reliable fashion. Cache databases, for example, ODI and HYPERSONIC SQL each implement both manual and automatic mechanisms to trim the cache size on low memory availability. Other cache management features are provided by server monitoring capabilities, such as the multiple database synchronization capabilities of Weblogic 6.1, as described earlier. If the cache database and the main database are both registered with the Weblogic 6.1 server (or other application servers having similar monitoring capabilities), the transactional capabilities of the server can be harnessed to manage synchronization of the cache database with the primary database in coordination with the CocoBase system and fully implement the cache plug-in features of CocoBase for enhanced system performance.

Optional Mapping Server Definitions and CocoBase System Integration

The CocoBase Optional Mapping Server is designed to integrate into and interoperate with standard Application Servers whose users need to have database access. The CocoBase runtime and Optional Mapping Server can be used to provide persistence and mapping functionality to the overall system irrespective of the type of object programming application server that is chosen to be utilized for a particular computer system.

Generally speaking, application servers enable corporations to develop, deploy, and manage business-critical applications that frequently need to have transactional coordination capabilities. Object programming applications that run under such servers can delegate certain functions, such as the transactional coordination to the server.

Many sound reasons exist for developing applications that can run under such servers and can utilize the CocoBase database access features for development and deployment. A few of those reasons are high performance, rapid development, scalability, easy integration with existing enterprise systems, state management, load balancing, and table/row locking mechanisms. Such features can be the keys to good html and Jave Server Pages (JSP) creation that can interface well with a user and permit the user to access to data and to conduct transactions that implement important features of

the server system and the CocoBase system that are invisible to the user. Examples are the server transactional coordination and the CocoBase runtime and caching systems. Such software application programs can be developed to delegate the functions of database access and data persistence to the CocoBase system, without the need to include database specific or persistence specific code in the application program source code. Moreover, CocoBase can automatically generate the server connectivity source code and persistence source code from the database maps.

CocoBase leverages the functionality of the Optional Mapping Server and features of application servers to provide additional advance mapping functions—typically the weakest link in Application Servers. CocoBase has become the pre-eminent mapping technology in Java, and provides superior mapping that can be readily utilized with virtually any mapping server as part of the overall system design. Application servers should be thought of as very complimentary technology to the CocoBase database mapping features, the CocoBase mapping server, CocoBase cache plug-in features, CocoBase runtime SQL generation facilities as well as other functionalities of the CocoBase software system product line.

Connection Pooling Systems with the Optional CocoBase Mapping Server

Connection pooling may be described as the concept of keeping database connections open and re-using them with new clients. Since it can be a resource intensive prospect to open a new database connection it is often advantageous to reuse existing database connections when possible, and to even use cached objects that are obtained from an earlier access of the database. A connection pool can be created for multiple clients. It keeps the connection to the database open for future requests. Creating a pool of JDBC connections gives clients ready access to connections that are already open. Connection pooling can improve performance by removing the overhead of opening and closing a new connection for each database operation or each time a new user tries to access the same database.

The connection management service of CocoBase can determine whether a database query can use a connection from the connection pool based on data source, user name, password, and the logged-on user account, etc. The connections in the pool can be shared among any users who can successfully match the login requirements. With this system design, each member client can get a connection from the pool for use, and then return it to the pool when finished (but the connection itself is not closed).

To configure the connection pooling of the CocoBase system, the developer can simply edit the Pool configuration file to set the desire connection pooling requirements (the file Cocopoolconfig.properties). Both the CocoServerSampleConnectionPool.java example plug-in and the CocoServerSamplecachePool.java example plug-in are designed to utilize this property file in order to configure the system Connection Pooling. The source code of each Connection Pooling plug-in can be readily customized to meet any project requirement not covered automatically in view of the discussion and description provided herein.

The CocoBase system runtime, data caching, object caching, and mapping features can be readily utilized by a developer to achieve a very scalable and high-performance data access and mapping system. This is accomplished by minimizing I/O between the client and server, in a fully multi-thread multi server architecture. The developer can permit client applications to decide when and where and how much data is sent between the client and server. This means the system does not need to burden down data pipes with irrelevant information or with lots of small independent network requests. Because less I/O is used, a system can be designed which utilize the same data bandwidth to support many more users with the CocoBase system implemented than it could have supported in the absence of CocoBase.

The present invention also provides a simple wrapper or facade interface and API to be utilized to coordinate relational data systems and object data systems that are described above and to control multiple software modules that are utilized to load, insert, update and delete data from data systems and objects or object models in object programming systems during implementation of such functions in the overall system or network. In a preferred aspect, the present invention provides a simple facade and object programming API designed to control the CocoBase O/R Software tools program components, particularly controlling the more complex components, to provide coordinated object graft and data persistence, including object model navigator components, transaction coordination components, CocoAdmin function components, and optionally caching components.

In a preferred aspect the simple wrapper or facade is called CBFacade and can be registered as a plug-in for the CocoBase O/R software tools as previously described. Some questions and answers regarding the functions of CBFacade are set forth below.

1) What is the connection property "cocosource.autoclose" for?

The "cocosource.autoclose" will force a connection/disconnection of the internal CocoDriver for each CBfacade call (this is ideal for connection pool based architectures—to use it, make sure "cocosource.url" contains your jdbc connection pool url settings).

2) What is the connection property cocosource.autotrans=true for?

What cocosource.autotrans=true does is simply tell the jdbc connection to function in autocommit mode (it corresponds to something like myjava_sql_connection.setAutoCommit(true)). It means it won't rely on a transaction manager for jdbc commits and it will set the jdbc layer to operate in autocommit mode. Therefore, each update/insert/delete at "row" level will be committed. This should not be set to true if you intend to use the CBFacade remote implementation (where the server portion runs inside an app server).

3) What is the connection property cocosource.autotidentity=true for?

This property should only be used with CS_fields. CS_fields are special map fields that cope with sequence maps to populate object field values upon insertion (see programmer's guide for more info on sequence maps), like when a database sequence is used for primary key generation. What this property actually does is to set a cocobase transaction not to buffer insert calls, but send them directly to the jdbc layer instead. This way, an new object can have its primary key fields populated right away and be able to establish database key relationships with linked objects.

4) What happens when beginTransaction( ) is invoked?

When CBFacade.beginTransaction( ) is invoked, a cocobase transaction object is set to start buffering changes in memory—the runtime won't talk to the jdbc layer at all. Only upon CBFacade.commit( ) the "memory" transaction is then committed and all buffered changes are propagated to the jdbc layer.

5) Is it required that beginTransaction( ) is called for each commit( )?

No, each save( )/delete( ) operation can be committed in separate by invoking commit( ) after each call like this:

```
...
myBase.save(myObj);
myBase.commit( );
...
```

6) An exception is thrown when two threads try to manipulate the same record—when the first has manipulated the record, the second catches the exception with an error code. What are the possible error codes and what do they mean?

CBFacadeException has four possible error codes:

CB_DELETE_ERROR (when performing a delete( )) or CB_SAVE_ERROR (when performing a save( )),

CB_INVALIDSTATE_ERROR and CB_UNKNOWN_ERROR.

CB_DELETE_ERROR and CB_SAVE_ERROR indicate that the object graph is out of date at some point, and the delete( )/save( ) procedure cannot be completed because of that. There is no way to know what specifically caused the error, but for sure some part of the graph was altered by other CBFacade instance (either in the same or in a different thread). When that happens, the client application has to "reload" the graph by calling load( ) again. This is optimized so that only out-of-date parts of the graph will be reloaded. The application has to explicitly load the graph. This is not done automatically by design. To enforce save/delete to always work, you can do something like:

```
while (true)
try {
    myBase.save(o);
    break; // save succeeded; exit loop
} catch (CBFacadeException e) {
    if (e.getCode( ) == CB_SAVE_ERROR) {
        o = myBase.load(o); // this will reload the graph
    } else throw e;
}
```

CB_INVALIDSTATE_ERROR means the whole CBFacade instance is compromised and there is no way to bring the object graph back to a consistent state (this can happen for instance in situations where the graph refers to an object that does not exist anymore in the database; it was somehow removed by another process that accessed the database directly, like through a sql client). In that situation, the application needs to call reset( ), which will clear all runtime cache and object graph contents.

CB_UNKNOWN_ERROR means that the runtime cannot to identify the nature of the error.

7) Is it possible to have multiple instances of CBFacade?

It is ok to use any number of CBFacade instances, but one should try to use the same instance for the same logical task or transaction. Scenarios where an application code uses CBFacade A for loading an object graph and CBFacade B for saving that same object graph should be avoided.

8) How does the CBFacade handle its JDBC connection?

The jdbc connection is handled by a CocoDriverInterface implementor object (i.e. CocoDriver), which is a CBFacade component. The CocoDriver acquires a jdbc connection from part of the url sent in the property "cocosource.url".

9) How does CBFacade handle object identity?

All object graph management is handled by a Navigator object, which is also responsible for assuring that only one instance with a given identity will exist in the graph pool it manages. The Navigator detects when instances are in fact the same logical object by comparing it primary key values whenever possible. Primary key fields are obtained from runtime CocoBase maps or from the Navigation model. If for some reason it cannot obtain the values for the primary key fields of an instance, the runtime will invoke equals( ) to compare that instance with other instances in the graph.

It is always a good idea to have the equals( ) method implemented based on primary key fields as a fall back mechanism but note that in this case the runtime will perform an exhaustive search among all objects in the graph pool. On the other hand, if the runtime can obtain the primary key values for an instance, it will use fast access hash based structures to detect if a copy is already in the graph.

The implementation of equals( ) is no longer a requirement, assuming the Navigator will be able to obtain key values. However, if the reconciliation mechanism falls back to the default Object.equals( ), problems may arise in object graph that contain cycles (which would not be detected upon loading) or if different copies of the same instance are added to the graph (this certainly happens in a distributed environment when objects are serialized/deserialized, for example, if you are using a CBFacadeRemote instance connected to the runtime through a generic session bean running in an app server). CocoAdmin generates code for the equals( ) method based on selected primary key fields accordingly.

10) Where is the dependency between the CBFacade and the object it can handle?

The only dependency is that, once an object is loaded or saved, it is bound to the memory graph/transaction held by that CBFacade instance. Only one instance with a given identity is kept in the object graph. Contents of copies of that same instance are reconciled accordingly upon each call (see answer to question 9).

11) If we don't use the CBFacade, what other runtime classes are available? Can we use other classes instead of CBFacade?

The CBFacade sources can be found under coco_home/demos/pguide/cbfacade—the sources are shipped with CocoBase so it is possible to see how other CocoBase runtime apis (e.g. Navigator, CocoProxyM, CocoDriverInterface, Transaction, CocoFactoryInterface, etc.) should be used for transparent persistence. However, we recommend the use of CBFacade for it is a simpler and higher level, already set up properly to work with object models of any complexity.

12) The CBFacade seems to handle references to objects. What consequences does this have related to garbage collection? Does it hold weak references?

At the moment we don't rely on weak references (this is not a simple matter: a graph can be loaded in the server jvm when the facade is running distributedly, so it is very likely there will be no references in that same jvm—the garbage collector would clear the graph pool), but we plan to in the future.

Internal references to linked objects cease to exist as objects are deleted from the graph (that means it has to have all its links removed; it doesn't necessarily mean objects must be deleted from the db).

Object graphs are not duplicated among several instances of CBFacade; each client app only loads the parts of the graph it is going to work with (as they would do with or without CocoBase), thus not being too memory intensive. Common graph parts (of two or more instances of CBFacade) are kept in synch by a notification mechanism. The CBFacade.reset( ) method will clear all the internal graph references and allow garbage collection to reclaim referred objects, in case one wishes to clear the graph object pool.

13) Can one CBFacade instance bind two or more objects? Say we load Obj1 with CBFacade, then load Obj2 and now want to update Obj1. Are there any problems with this?

CBFacade is made to work with connected or disconnected graphs of objects. There is no problem in using it to manage graph obj1 and graph obj2. If same object should appears in both graphs (which means they are connected), it will assure one single instance is shared and reachable from both graph roots obj1 and obj2, thus preserving identity of the objects.

14) Is CBFacade stateless? What information is hold by CBFacade?

CBFacade, when used distributedly, connects to a stateful session bean; that means: each client app has its own copy of the graph running in the app server—copies are notified of inconsistencies when they happen and a CBFacadeException (application exception) is sent back to the client which can then take the appropriate action for that situation (for instance, if a graph is invalidated by other CBFacade instance, that graph can be "revalidated" by calling load( ) again). The same behavior is verified in 2-tier multi-threaded apps. This is by design—in most cases, it is better for the client app to decide what to do when such conflicts occur (very useful in optimistic locking scenarios, for instance).

15) Say there are two threads T1 and T2 working on one facade instance. Does CBFacade can handle correct: T1: load Obj1; T2: load Obj2; T1: save Obj1; T2: save Obj2

The facade api is not synchronized neither are its components. The best approach is to have one CBFacade instance for each thread. Different CBFacade instances managed by distinct threads/clients are notified of inconsistencies changes in object graph. The client thread can then handle that situation accordingly.

In ideal scenarios, an instance of CBFacade is created/connected by a given thread, used in given task, and closed when that task is completed (and this will clear any references to cached/bound objects). The other option is to create a synchronized wrapper over the CBFacade api, but we don't recommend that as it may become slow and too memory intensive (imagine a huge graph of objects loaded in memory by that singleton facade instance; performance is proportional to the size of the graph).

16) What happens if the user works with the app and loads 100 different objects? Are there 100 different objects cached/bound in memory?

Yes, there will be. A copy of the entire graph of objects is kept in memory so that changes in the graph can be reconciled efficiently, without requiring database access. An alternative to prevent unnecessary loading would be to use "lazy" load. In that case, objects would be brought to memory as links are navigated by the user ("lazy" load can be easily set up later during development process by changing the navigation model; see programmer's guide for more details).

17) What happens, if the thread that holds a facade instance dies? Are the CBFacade instance and cached/bound objects garbage collected?

Yes, everything will be garbage collected as long as the thread that died was the only one to refer to the CBFacade instance.

18) I have more than one factory. How can I use CBFacade since it accepts only one factory name in "cocosource.factory"?

Multiple factories must be combined in a single factory to be used with the facade. One way to combine them is to have a combiner factory that simply delegate to the other factories like below:

```
public class AppFactory {
    CocoFactoryInterface factoryX = new FactoryX( );
    CocoFactoryInterface factoryY = new FactoryY( );
    ....
    CocoFactoryInterface factoryN = new FactoryN( );
    public Object getInstance(Object src, Object props, String
    objectName) {
        if (objectName = "Object X") {
            return factoryX.getInstance(src,props,objectName);
        else if (objectName = "Object Y") {
            return factoryY.getInstance(src,props,objectName);
        ...
        else if (objectName = "Object N") {
            return factoryN.getInstance(src,props,objectName);
    }
}
```

You can then register this "combined" AppFactory through the connection property "cocosource.factory".

19) Does CBFacade supports inheritance?

Yes, it supports inheritance through the factory mechanism. CocoBase runtime offers a generic factory that manages inheritance based on inheritance models. Simply register that factory in the "cocosource.factory" property. Also, set any factory specific properties like the name for the inheritance model and they will be send to the factory upon initialization. For example:

```
        Properties connectionProperties = new Properties( );
        connectionProperties.setProperty("cocosource.name","thought.CocoBase.CocoPowder");
        connectionProperties.setProperty("cocosource.url","jdbc:oracle:thin:@localhost:1521:orcl")
;
        connectionProperties.setProperty("cocosource.jdbcdriver","oracle.jdbc.driver.OracleDriver"
);
        connectionProperties.setProperty("cocosource.user","sa");
        connectionProperties.setProperty("cocosource.password","");
        connectionProperties.setProperty("cocosource.navmodel","MyNavModel");
        connectionProperties.setProperty("cocosource.factory","thought.CocoBase.InheritanceFact
ory");
```

-continued

```
// this will be send to the InheritanceFactory instance upon its initialization
connectionProperties.setProperty("cbfactory.inheritancemodel","MyInheritanceModel");
myBase.connect(connectionProperties);
```

Transparent Persistence with CocoBase

CocoBase accomplishes transparent persistence with java object models without using bytecode manipulation, proprietary interfaces or class hierarchy intrusion. This means that no special classes or interfaces are needed in the Object Model in order to do persistence with CocoBase. The only requirement is that they must have a default constructor with no arguments.

numberous other documents in the art, the CocoBase Navigator can easily develop a navigation model for the application. This model can then be used to persist the objects.

The Example below is not directed to persistence of the Navigation model for the above Cityinfo application, but the concepts can be easily applied to the Cityinfo application to provide transparent persistence.

First, open a CocoBase connection as follows:

```
CocoDriverInterface myBase = CocoDriver.getCocoDriver(
    "thought.CocoBase.CocoPowder",
    "org.hsql.jdbcDriver",
    "jdbc:HypersonicSQL:hsql://localhost;cocoprop=cocofactory=CocoProxyFactory",
    "sa", ""
)
if (myBase.connect( ) == -1) {
    System.out.println("Failed connect!");
    System.exit(1);
}
...
```

There are 3 basic Runtime components that are involved in the transparent persistence of objects:

- The CocoBase Runtime O/R mapping class that wrappers the JDBC driver and issues queries and does the actual persistence calls. This is a class such as thought CocoBase CocoPowder or thought CocoBase.CocoPowderPlugin20 (for jdbc 2.0 connections).
- The thought.CocoBase.Transaction object that can track changes of instances, and acts as a change 'buffer'. If a Transaction object is used, then it only calls a CocoBase Runtime driver—O/R mapping runtime class—when the txn.commit( ) is called.
- The thought.CocoBase.navapi.Navigator object that can track and detect changes in relationships based on Link definitions. Note that unlike CocoBase Maps, Link models are not kept in the CocoBase repository. Once a link definition model is created it is saved in the demos/resources directory (by default—although this can be overridden). As long as the model is in the classpath either directly or in a subdirectory called 'resources' it will find the model properties file and retrieve the navigation information.

The Navigator class can function in conjunction with a Transaction object or it can function standalone. While complex object graphs can be transparently managed directly by the Navigator and without the Transaction object, the use of the Transaction object is generally preferred because of its buffering and update optimizations which only persist those attributes that have changed.

Creating Applications with CocoBase Transparent Persistence

After classes have been generated and compiled, such as with the above application, CocoBase runtime classes can be used to persist instances of these classes. As shown in

Now create a CocoBase transaction object to manage any objects that are retrieved. Notice how the transaction object is configured through parameters and property settings.

```
thought.CocoBase.Transaction cocoTxn =
    new thought.CocoBase.Transaction(myBase, false);
Properties cocoTransProps = new Properties( );
cocoTransProps.put ("preserveCommit", "true");
cocoTransProps.put ("commitconnection","true");
cocoTransProps.put ("throwExceptions","true");
cocoTransProps.put ("updateOnlyChangedColumns","true");
cocoTxn.setProperties(cocoTransProps);
// Begin a new transaction.
cocoTxn.begin( );
```

Then open a CocoBase Navigator object with the Navigation model to be used and also we register the Transaction object with the Navigation model:

```
// Instantiate a Navigator with the Link model created from
// the UML/XMI document
thought.CocoBase.navapi.Navigator navigator =
    new thought.CocoBase.navapi.Navigator (myBase, "company");
// Assign the current transaction to the Navigator
navigator.setTransaction(cocoTxn);
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
// Setup our object to query
Department dept = new Department( );
department.setName("SALES");
// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = myBase.selectAll(
                        new thought.CocoBase.CocoProxyM(dept),"Department");
```

The CocoProxyM is optional, the code example below shows the same call without its usage.

```
// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = myBase.selectAll(dept, "Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the loadAllLinks method which does the automatic navigation for that object:

```
for(int i=0; i< deptVector.size( ); i++) {
    Department d = (Department)deptVector.elementAt(i);
    // Because the cascadeLoad flag is set to true in the direction
    // Department->Employees, the employees link will load
    automatically
    d = navigator.loadAllLinks (d, "Department");
    ...
    Vector emps = d.getEmployees( );
    for (int j=0; j<emps.size( ); j++) {
        // raise salaries by 20%
        emp.setSalary(emp.getSalary( ) *1.2);
        ...
    }
    ...
    // Once changes are made to an object graph those changes can be
    // synchronized using the updateAllLinks method such as:
    navigator.updateAllLinks (d, "Department", true);
}
```

You can then commit the buffered changes with:

cocoTxn.commit( );

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-J2EE applications the application intrusion is incredibly small, requiring a single method call for each root node.

The Navigator supports one-to-one, one-to-many and many-to-many relationships with cycle detection. It also detects this locally (i.e. in the client application) or by reconciling serialized or copied objects without any kind object model or bytecode intrusion. This is truly transparent persistence that is architected and designed for the VM oriented Java language. There are also more advanced applications included in the demos/pguide/navapi subdirectory that demonstrate one-to-one, one-to-many and many-to-many relationships as well as an EJB using the Navigator system to manage a graph of java objects.

The Example below is not directed to persistence of the Navigation model for the above Cityinfo application, but the concepts can be easily applied to the Cityinfo application to provide transparent persistence.

First, open a CocoBase CBFacade Transparent Persistence connection as follows:

```
// Set up the connection class - can be Local or Remote with EJB Server
CBFacade cbf =
CBFacade.create("thought.CocoBase.CBFacadeLocal");
Properties props = new Properties();
props.put("cocosource.name",args[0]);
props.put("cocosource.jdbcdriver",args[1]);
props.put("cocosource.url",args[2]);
props.put("cocosource.user",args[3]);
props.put("cocosource.password",args[4]);
props.put("cocosource.navmodel","CompanyAppLinks");
cbf.connect(props);
...
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
// Setup our object to query
Department dept = new Department();
department.setName("SALES");
// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = cbf.loadAll(dept,"Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the loadAll method which does the automatic navigation for that object:

```
for(int i=0; i< deptVector.size(); i++) {
    Department d = (Department)deptVector.elementAt(i);
    // Because the cascadeLoad flag is set to true in the direction
    // Department->Employees, the employees link will load automatically
    d = cbf.loadAll(d,"Department");
    ...
    Vector emps = d.getEmployees();
    for (int j=0; j<emps.size(); j++) {
        // raise salaries by .20%
        emp.setSalary(emp.getSalary()*1.2);
        ...
    }
    ...
```

-continued

```
        // Once changes are made to an object graph those changes can be
        // synchronized using the save method such as:
        cbf.save(d,"Department");
    }
```

You can then commit the buffered changes with:

cbf.commit( );

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-J2EE applications the application intrusion is incredibly small, requiring a single method call for each root node.

Both the above architecture and two implementations presented herein are uniquely suited to work in every application from the tiny local app to the enterprise J2EE and to do so with superior performance and manageability.

Additionally, by using the JMS (or similar) notification caching system described herein with the persistence systems described above persistence can be distributed while maintaining a real-time caching system. When an application or multiple users need to do a lot of data reads, without a lot of data writes while maintaining database integrity, such a caching system can yield enhanced performances of distributed applications with database persistence.

While a number of preferred embodiments of the present invention have been discussed by way of example, the present invention is not to be limited thereby. Rather, the present invention is to be construed as including any and all variations, modifications, permutations, adaptations and embodiments that would occur to one skilled in this art once having been taught the present invention.

We claim:

1. A computer system or network comprising a local or distributed data or object caching system wherein:

(a) data or objects are (i) placed in a user cache in the caching system when the data or object is first read from a data source by a user as a lazy cache, or (ii) preloaded as directed by a user,

(b) each unit of data or object that is placed in a user cache is registered with a unit notification system (UNS) as being present in the cache and the UNS is connected to a Java Messaging Service (JMS) or to any other reliable messaging service available in the object programming language system environment,

(c) when any unit of data or object is updated in a user caches and persisted (saved) to a primary data source the UNS sends a message to all other user caches that may exist and contain the updated unit of data or object to register in the user cache a message that the unit was updated by another user and is noted as being invalid in the cache, and

(d) other user caches that have been notified as described in (c) will read from the data source the updated unit of data or object and update the cache only when a user attempts to access the invalidated unit or data or object in a user cache, may automatically update the user cache as soon as notified as has been pre-set in the caching system by the user of the user cache.

2. The computer system or network according to claim **1**, comprising a local object or data caching system wherein multiple applications or users access the data in an object language virtual machine environment.

3. The computer system or network according to claim **2**, comprising a distributed object or data caching system that may be distributed over an intranet or internet environment and may include distributed users, distributed components, distributed caches, multiple cache copies, or any combination thereof wherein a central data source is used for insertions, deletions, or changes to the cached data or objects.

4. The computer system of claim **3**, wherein the system also provides transparent persistence.

5. The computer system of claim **2**, wherein the system also provides transparent persistence.

6. The computer system of claim **1**, wherein the system also provides transparent persistence.

7. The computer system of claim **1**, wherein the system further comprises at least one Object to Relational Mapping Repository.

8. A software module or modules for an object programming application comprising a local or distributed data or object caching component, or components, wherein the:

(a) the software component, or components, provide that data or objects are (i) placed in a user cache in the caching system when the data or object is first read from a data source by a user as a lazy cache, or (ii) preloaded as directed by a user,

(b) the software component, or components, provide the logic for placing each unit of data or object in a user cache and registering the unit or object with a unit notification system (UNS) as being present in the cache and the for connecting the UNS to a Java Messaging Service (JMS) or to any other reliable messaging service available in the object programming language system environment, and

(c) the software components, or components, containing logic providing that when any unit of data or object is updated in a user caches and persisted (saved) to a primary data source the UNS sends a message to all other user caches that may exist and contain the updated unit of data or object to register in the user cache a message that the unit was updated by another user and is noted as being invalid in their cache as an unreliable data unit or object.

9. The software module, or modules, of claim **8**, wherein the software component, or components, further provide the logic directing other user caches that have been notified that a data unit or object has been updated to only read from the data source an updated unit of data or object to update their user cache when the invalidated unit of data or object in their user cache needs to be accessed, or provides that their cache may automatically be updated as soon as notified as has been pre-set in software by the user of the user cache.

10. The software module, or modules, according to claim **9**, comprising logic to access a local object or data caching system wherein multiple applications or users access the data in an object language virtual machine environment.

11. The software module, or modules, of claim **10**, wherein the software also provides transparent persistence.

12. The software module, or modules, according to claim **9**, comprising logic to access a distributed object or data caching system that may be distributed over an intranet or internet environment and may include distributed users, distributed components, distributed caches, multiple cache copies, or any combination thereof wherein one or more central data sources are used for persisting changes to the cached data or objects.

13. The software module, or modules, of claim **12**, wherein the software also provides transparent persistence.

14. The software module, or modules, of claim **9**, wherein the software also provides transparent persistence.

15. The software module, or modules, of claim **9**, wherein the software also provides a component or class for communicating with at least one Object to Relational Mapping Repository.

16. The software module, or modules, of claim **8**, wherein the software also provides transparent persistence.

* * * * *

EXHIBIT C

US006999956B2

(12) **United States Patent**
Mullins

(10) **Patent No.: US 6,999,956 B2**
(45) **Date of Patent: Feb. 14, 2006**

(54) **DYNAMIC OBJECT-DRIVEN DATABASE MANIPULATION AND MAPPING SYSTEM**

(76) Inventor: **Ward Mullins**, 657 Mission St., San Francisco, CA (US) 94105

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 568 days.

(21) Appl. No.: **09/987,489**

(22) Filed: **Nov. 15, 2001**

(65) **Prior Publication Data**

US 2002/0091702 A1 Jul. 11, 2002

Related U.S. Application Data

(60) Provisional application No. 60/249,418, filed on Nov. 16, 2000.

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/2**; 707/3

(58) **Field of Classification Search** 707/1, 707/3, 4, 5, 10, 100, 102, 104.1, 2; 345/736; 711/141; 709/201, 203, 225, 251; 713/201; 717/108, 114, 162, 165

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,291,583	A	3/1994	Bapat	707/104.1
5,426,780	A	6/1995	Gerull et al.	707/3
5,966,707	A	10/1999	Van Huben et al.	707/10
6,134,559	A	10/2000	Brumme et al.	707/103 R
6,163,776	A	12/2000	Periwal	707/4
6,754,670	B1 *	6/2004	Lindsay et al.	707/100
2002/0120603	A1 *	8/2002	Thompson	707/1
2003/0055826	A1 *	3/2003	Graham	707/10
2003/0163460	A1 *	8/2003	Shinn et al.	707/3
2003/0208505	A1 *	11/2003	Mullins et al.	707/102

OTHER PUBLICATIONS

Susan D. Urban and Taoufik Ben Abdellatif, "An Object–Oriented Query Language Interface to Relational Database in a Multidatabase Database Environment", Department of Computer Science and Engineering, Arizona State University.

Tanisi Pereira de Carvalho, Nina Edelweiss, "A Visual Query System Implementing a Temporal Object–Oriented Model with Roles on a Relational Database", Instituto de Informatica, Universidade Federal do Rio Grande do Sul, Porto Alegre—RS—Brazil.

Jean–Marc Nerson, "Applying Object–Oriented Analysis and Design", Communications of the ACM/Sep. 1992/vol. 35, No. 9.

Marco Baldassari, Giorgio Bruno, "An Environment for Operational Software Engineering in Ada", Dipartimento di Automatica e Informatica, Italy.

Richard Hull, "Managing Semantic Heterogeneity in Database: A Theoretical Perspective", Bell Laboratories, New Jersey.

Witold Litwin, France, Leo Mark, and Nick Roussopoulos, Department of Computer Science, University of Maryland, Maryland, "Interoperability of Multiple Autonomous Database", ACM Computing Surveys, vol. 22, No. 3, Sep. 1990.

* cited by examiner

Primary Examiner—Apu M. Mofiz
(74) *Attorney, Agent, or Firm*—Robert Lev

(57) **ABSTRACT**

The present invention provides a system and method for dynamic object-driven database manipulation and mapping system which relates in general to correlating or translating one type of database to another type of database or to an object programming application. Correlating or translating involves relational to object translation, object to object translation, relational to relational, or a combination of the above. Thus, the present invention is directed to dynamic mapping of databases to selected objects. Also provided are systems and methods that optionally include caching components, security features, data migration facilities, and components for reading, writing, interpreting and manipulating XML and XMI data files.

12 Claims, 7 Drawing Sheets

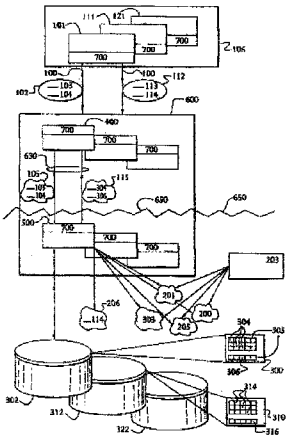


Figure 1

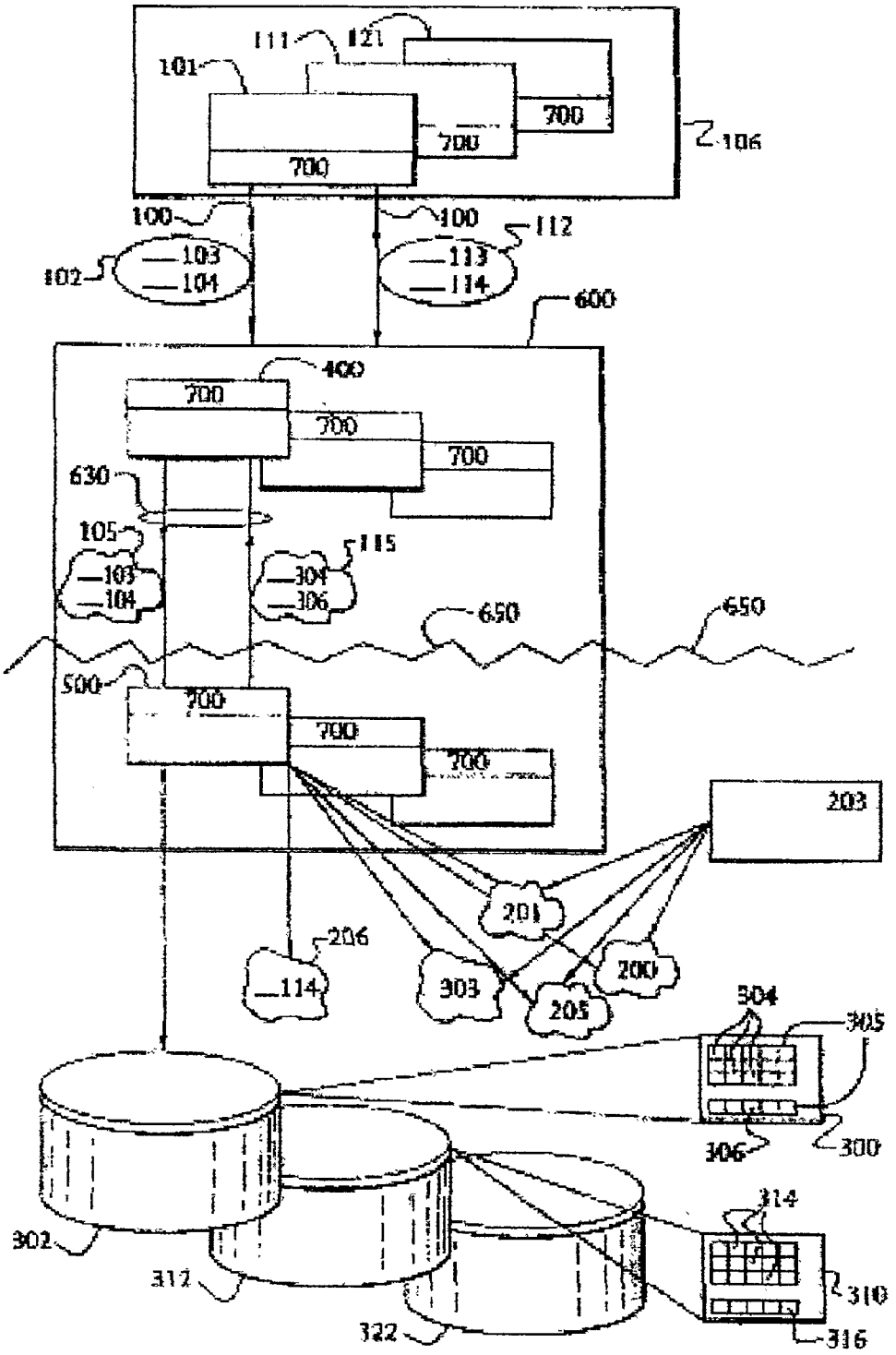

111
121
101
700
700
700
106
100
100
112
102
103
104
113
114
600
400
700
700
700
630
105
103
104
115
304
306
650
650
500
700
700
700
203
201
206
114
303
200
205
304
305
306
300
302
312
314
310
322
316

Figure 2

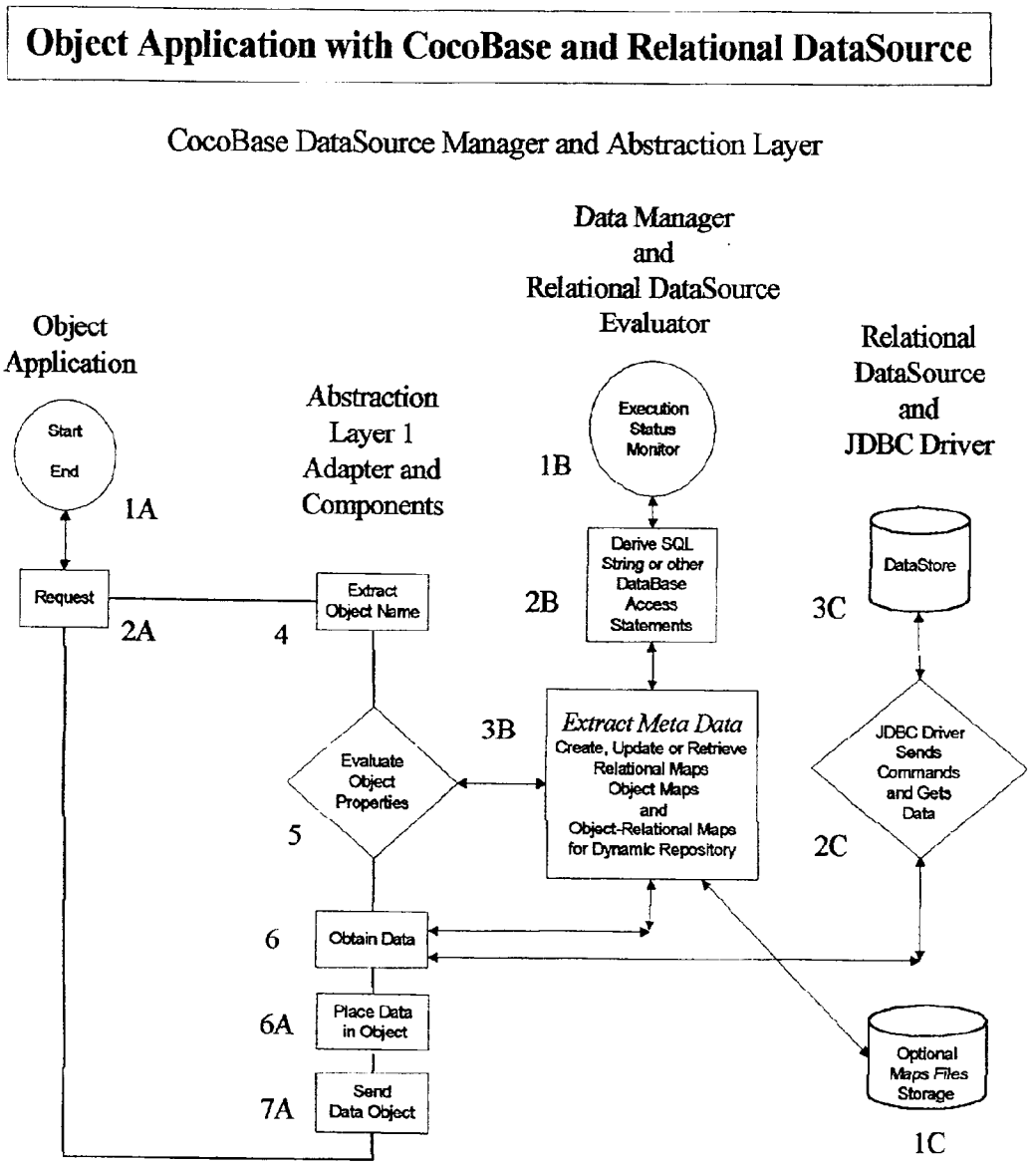

Object Application with CocoBase and Relational DataSource
CocoBase DataSource Manager and Abstraction Layer
Data Manager and Relational DataSource Evaluator
Object Application
Abstraction Layer 1 Adapter and Components
Relational DataSource and JDBC Driver
Start
End
1A
Execution Status Monitor
1B
Request
2A
Extract Object Name
4
Derive SQL String or other DataBase Access Statements
2B
DataStore
3C
Evaluate Object Properties
5
3B
Extract Meta Data
Create, Update or Retrieve Relational Maps Object Maps and Object-Relational Maps for Dynamic Repository
JDBC Driver Sends Commands and Gets Data
2C
6
Obtain Data
6A
Place Data in Object
7A
Send Data Object
Optional Maps Files Storage
1C

Figure 3

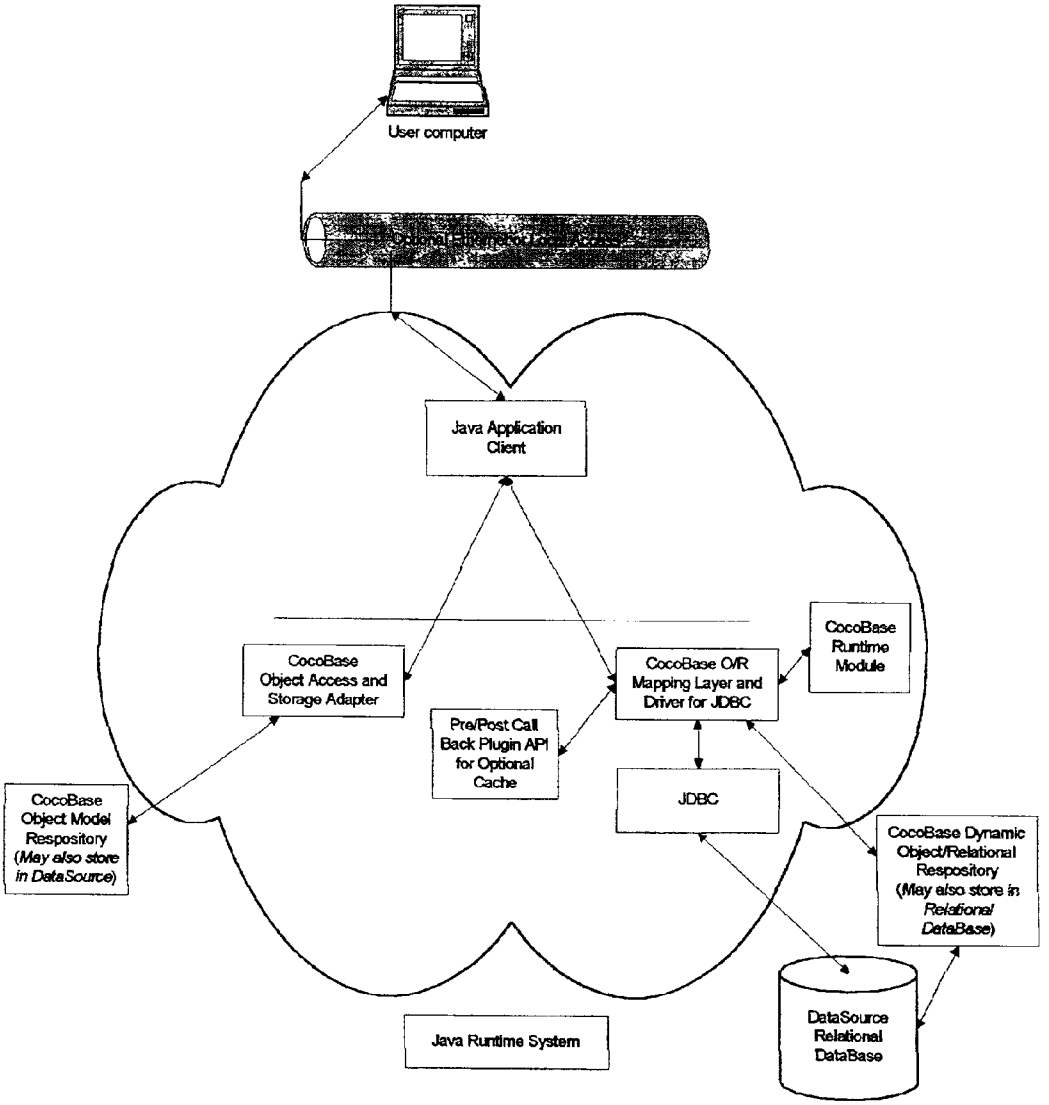

User computer
Java Application Client
CocoBase Object Access and Storage Adapter
CocoBase O/R Mapping Layer and Driver for JDBC
CocoBase Runtime Module
Pre/Post Call Back Plugin API for Optional Cache
JDBC
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
DataSource Relational DataBase
Java Runtime System

Figure 4

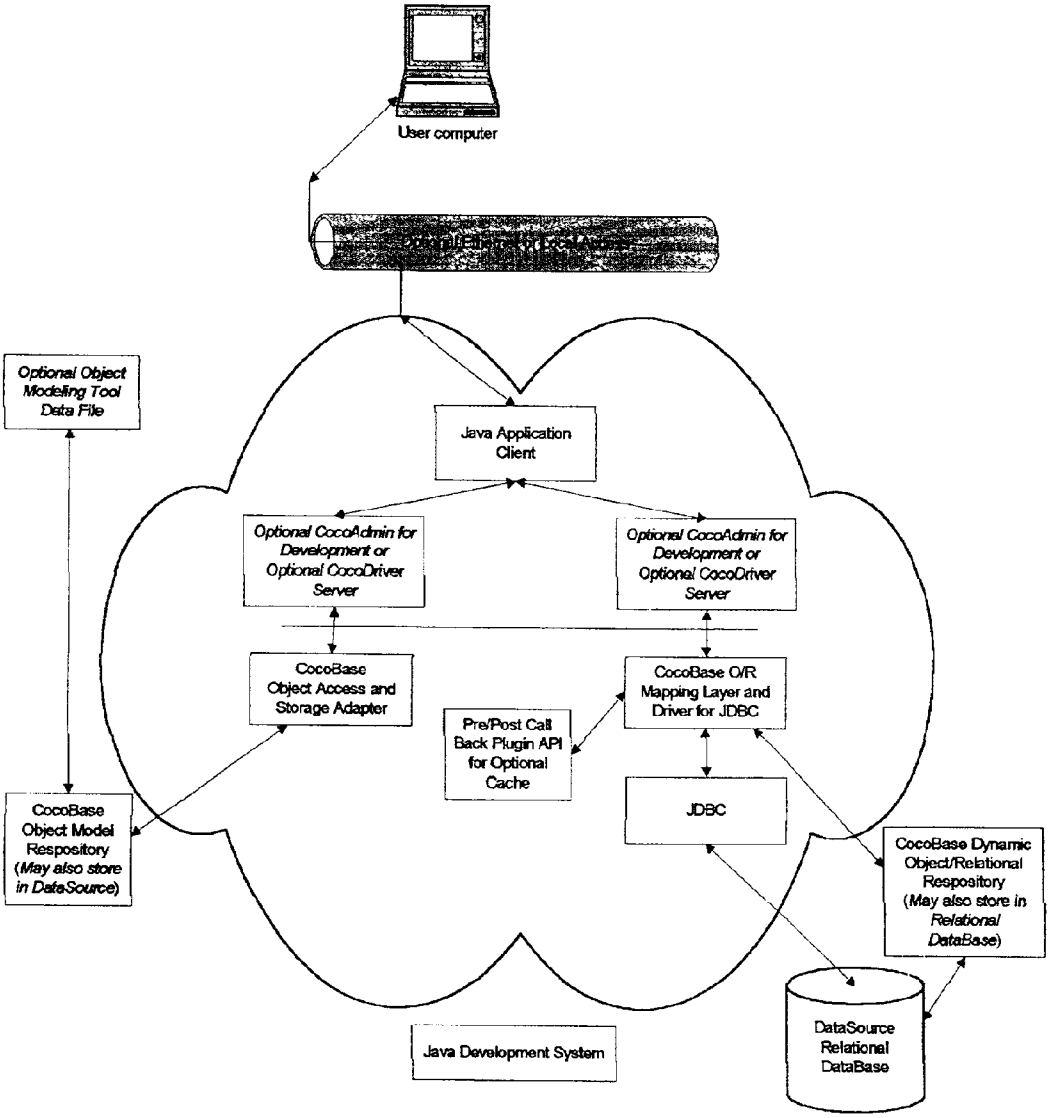

User computer
Optional Object Modeling Tool Data File
Java Application Client
Optional CocoAdmin for Development or Optional CocoDriver Server
Optional CocoAdmin for Development or Optional CocoDriver Server
CocoBase Object Access and Storage Adapter
CocoBase O/R Mapping Layer and Driver for JDBC
Pre/Post Call Back Plugin API for Optional Cache
JDBC
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
DataSource Relational DataBase
Java Development System

Figure 5

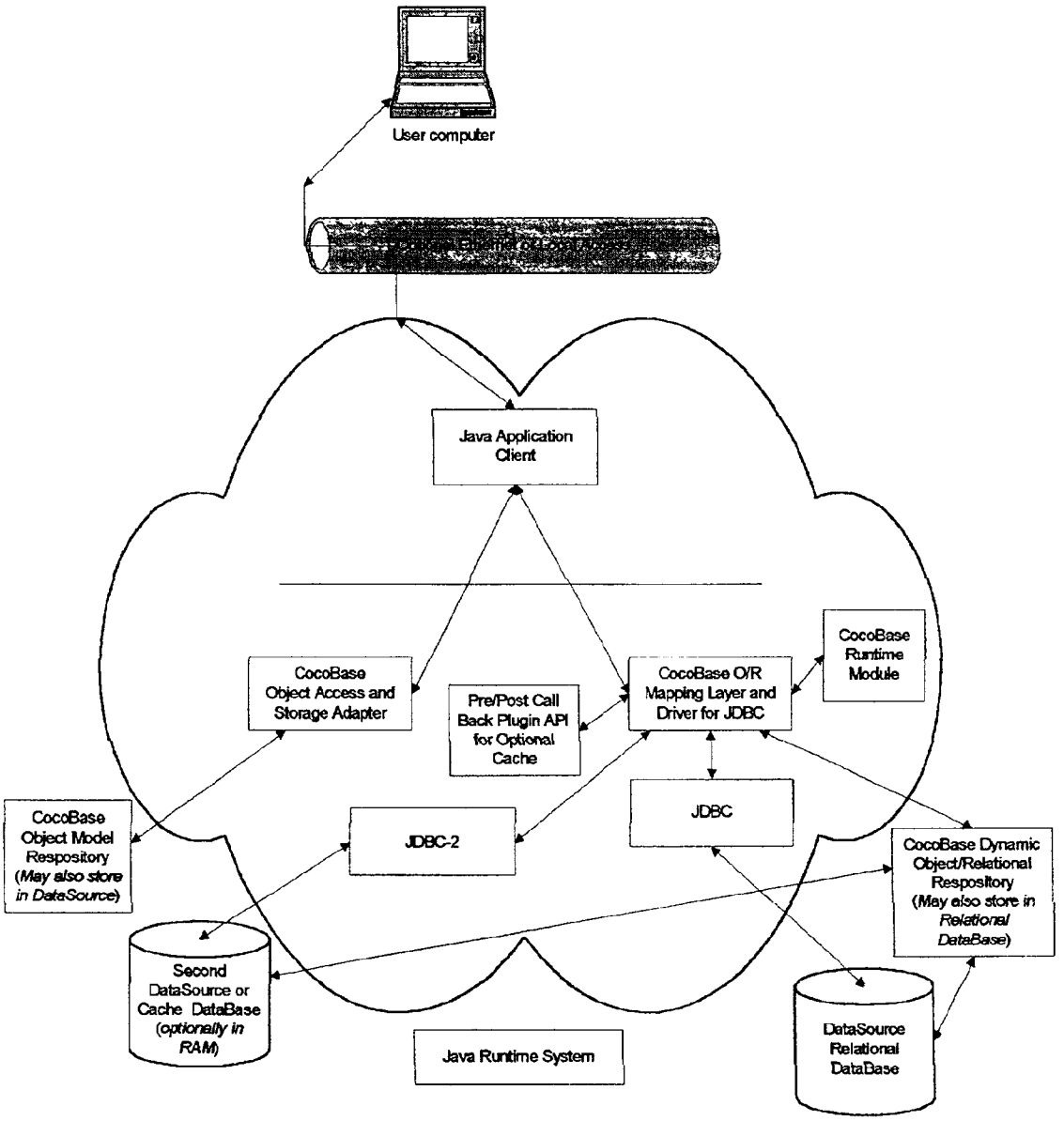

User computer
Java Application Client
CocoBase Object Access and Storage Adapter
Pre/Post Call Back Plugin API for Optional Cache
CocoBase O/R Mapping Layer and Driver for JDBC
CocoBase Runtime Module
JDBC
JDBC-2
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
Second DataSource or Cache DataBase (optionally in RAM)
Java Runtime System
DataSource Relational DataBase

Figure 6

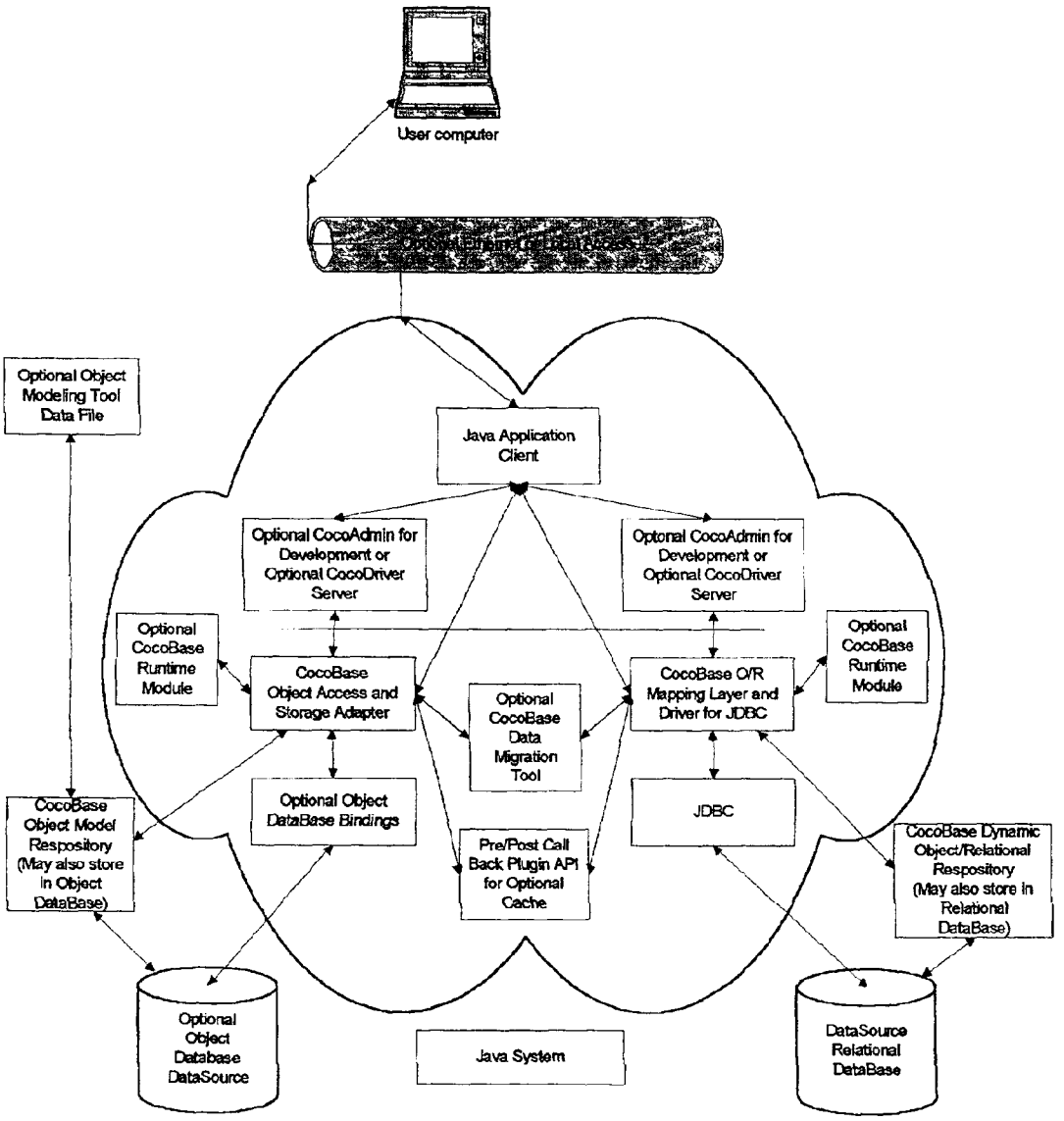
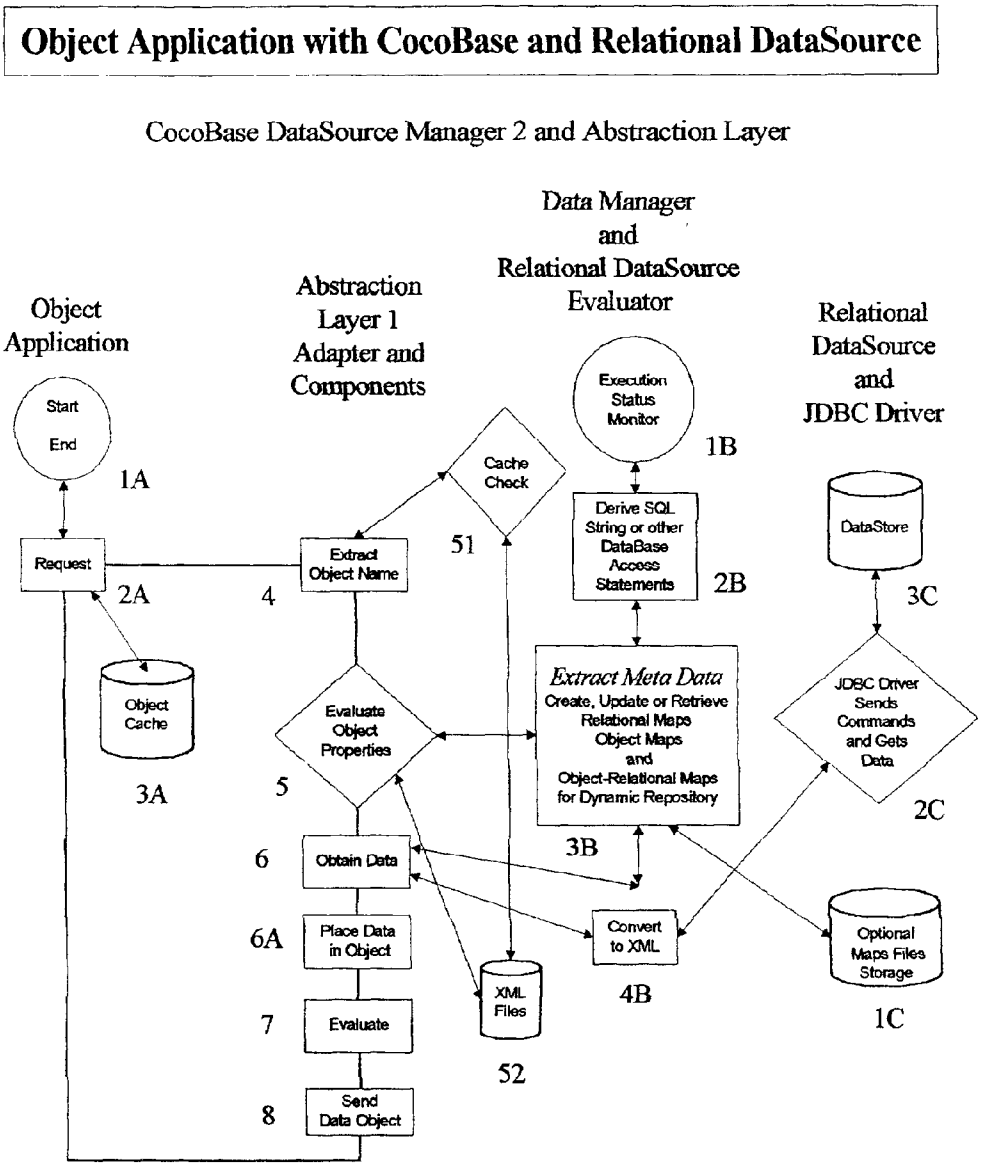

User computer
Optional Object Modeling Tool Data File
Java Application Client
Optional CocoAdmin for Development or Optional CocoDriver Server
Optonal CocoAdmin for Development or Optional CocoDriver Server
Optional CocoBase Runtime Module
CocoBase Object Access and Storage Adapter
Optional CocoBase Data Migration Tool
CocoBase O/R Mapping Layer and Driver for JDBC
Optional CocoBase Runtime Module
CocoBase Object Model Respository (May also store in Object DataBase)
Optional Object DataBase Bindings
Pre/Post Call Back Plugin API for Optional Cache
JDBC
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
Optional Object Database DataSource
Java System
DataSource Relational DataBase

Figure 7


Object Application with CocoBase and Relational DataSource
CocoBase DataSource Manager 2 and Abstraction Layer
Object Application
Abstraction Layer 1 Adapter and Components
Data Manager and Relational DataSource Evaluator
Relational DataSource and JDBC Driver
Start
End
1A
Request
2A
Object Cache
3A
Extract Object Name
4
Cache Check
51
Execution Status Monitor
1B
Derive SQL String or other DataBase Access Statements
2B
DataStore
3C
Evaluate Object Properties
5
Extract Meta Data
Create, Update or Retrieve Relational Maps Object Maps and Object-Relational Maps for Dynamic Repository
3B
JDBC Driver Sends Commands and Gets Data
2C
6
Obtain Data
6A
Place Data in Object
7
Evaluate
8
Send Data Object
XML Files
52
Convert to XML
4B
Optional Maps Files Storage
1C

DYNAMIC OBJECT-DRIVEN DATABASE MANIPULATION AND MAPPING SYSTEM

This continuation-in part application of U.S. patent application Ser. No. 60/249,418, filed Nov. 16, 2000, and a Provisional patent application Ser. No. 09/162,243 filed Oct. 25, 2001.

TECHNICAL FIELD

The present invention relates in general to correlating or translating one type of database to another type of database or to an object programming application. Correlating or translating involves relational to object translation, object to object translation, relational to relational, or a combination of the above. Thus, the present invention is directed to dynamic mapping of databases to selected objects.

BACKGROUND ART

Computer databases or similar constructions (hereinafter referred to as data stores) are powerful tools for storage, organization, retrieval and other handling of various types of information. However, there are different database models, or formats, for data access that are incompatible with each other, and may also be incompatible with, or remarkably different from, an object programming application. In this respect, complex relationships between objects present in the object programming application may be entirely absent in a relational or object database being accessed or updated. Nonetheless, a plurality of these database types have achieved a high level of popularity and proliferation. The two most common database models are object and relational, and object programming models are frequently used for internet applications involving database accessing.

As an example of common object programming applications, the popular Java language with its server-side component, Enterprise Java Beans, is being widely used to support object model programming applications that access and use data from databases that have a JDBC driver. Thus, it is desirable to be able to adapt and use many of the traditional relational database tools, utilities, systems and corporate procedures with the newer object format of many more recent web-based applications. Since many users may be accessing a single application, there is a need for speeding up database accesses and for synchronizing object programming application transactions with database accesses and updates.

Systems for accessing data storage based upon objects have been used for many years to accommodate object-directed software applications. The most common conventional approach to accomplish access of data stores involves writing and imbedding custom access codes within an object application needing access to the data. This approach is generally limited to having custom codes capable of accessing only a single relational table within a relational database or similar construction, referred to as a data store. When an object model has been adapted specifically for accessing a single data store using the JDBC driver for that data store, moving data to a second database or accessing multiple databases can become very problematic and painful for a programming engineer who is assigned the responsibility for such a task.

The aforementioned conventional approach to bridging between object and relational models (or between object and multiple relational models) has frequently required "brute-force" hand-recoding of relational database access code. This approach can be very time-consuming and is prone to

errors. This approach can also generate inefficient objects that require more processing time, storage space, and transfer time. Further, this approach does not scale well because each instance of relational access code must be converted manually, in turn, even if similar objects have already been applied for use with the converted relational data base.

As mentioned above, more efficient approaches sometimes use tools to automate the conversion process. However, the tools provided to date typically are not versatile enough to handle many types of conversion without manual coding or explicit defining of parameters by a human operator. Because of this requirement for operator participation in the translation, the conversion using today's tools still does not scale well and does not allow for fully automated and flexible conversion of data access between various data base models.

Further, this approach to conversions does not provide for communicating the conversions in an efficient manner as, for example, to other entities who may need the same type of conversion on similar data, or who may require the conversion to access the data in another way. In a distributed environment that may involve multiple users accessing multiple databases by multiple object applications, this can get exceeding complex. Another drawback of conventional systems and techniques (as understood in the conventional art), is that dynamic mapping of objects to multiple types of databases is virtually impossible, because the tailored hand code must be updated and recompiled. This may require that an application be stopped and restarted in order to implement mapping changes, i.e., the mapping changes are not dynamic.

Significant drawbacks are associated with using a relational database model in conjunction with an object programming model, but the relational database model is still the most widely accepted traditional format for reliable and quick access to data while using complex data search queries. This model allows the user to store data records and to access the data via a query language that uses relational expressions such as AND, OR, NOT, etc. Over the years, the vast majority of production databases have been relational databases, each having its own relational arrangement, and accessible by a particular query language. While convenient for set-up and arranging data, access is had only by those having substantial knowledge of the various relationships between pieces of data in the database.

JDBC drivers have been designed to formulate database specific SQL queries or statements from standard SQL strings passed to them by an object programming application. Ordinarily, such JDBC drivers are specific to a database, and separate statements and separate JDBC drivers must be utilized to access different databases. This is complicated by the fact that many object programming applications either do not permit simultaneous attachment to multiple databases or poorly coordinate transactions that impact multiple databases. Accordingly, relational databases can sometimes be awkward to access, and transferring data from one database to another database can be slow and painful, unless a specific transfer program has been specifically written or tailored to effect that specific transfer of data.

The object database model is a newer model that has rapidly increased in usage due to recent trends such as the world-wide-web and application service provider (ASP) architectures. Such object databases have sometimes been used as the primary source for accessing data, for example, Jasmine and ObjectStore are available from Computer Associates Intl., Inc. and Object Design Inc. (ODI), respectively.

However, accessing data in such permanently stored object database has raised serious performance issues for larger applications when complex querying is required (as compared to relational databases). Complex querying is usually faster using relational databases since query engines may be more highly developed for relational database accesses.

Previously, object applications have tended to only use simple memory buffers with relational and object databases. Difficulties existed with using either an object database or a relational database to serve as a transient memory resident database "cache" (secondary database) to try to speed up database accesses. No commercially viable solutions have existed with acceptable performance.

There is a need for an object programming application to successfully access a secondary transient memory resident database, as either a relational database or an object database. In such a case, a dynamic database mapping tool would be necessary to provide the object programming application with a map or maps to both the primary and secondary databases in order to successfully implement a dynamic transient memory resident database. It would be particularly helpful in optimizing the performance of an object programming application if there could be provide a memory resident object database (or a resident portion thereof) as a cache for data retrieval. In such a case it might be possible to directly access (and perhaps re-use) data objects stored in the secondary transient resident memory database, including their metadata, instead of generating data objects from accessed data.

If it were possible to successfully provide a commercially viable secondary memory resident transient database, data being used by the object programming application would only need to be persisted back to one or more of a permanent storage relational database (primary database) when data has been created, inserted or updated. Ideally, the application would likewise update the memory resident object database (secondary database) when the primary database is updated. However, as mentioned such simultaneous access of multiple databases can give rise to transactional coordination problems, or to data synchronization problems.

Accordingly, intensely needed by many modern software application programs are systems for more efficient access and manipulation of data stores, systems having the flexibility and dynamic capability to attach data from a database to maps as objects and having the ability to map one or more databases to various objects in real time. A strong need exists for such systems that also permit a user to cleanly, transparently and synchronically transfer data between multiple data sources, while maintaining the ability for an object programming application to access or use such data in the system. The need exists for an improved database access buffering system, for object programming applications, having at least one transparent secondary memory resident database and a primary data source that can be simultaneously utilized in a synchronized and transaction coordinated manner.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the present invention to overcome the drawbacks of the conventional art and achieve flexibility and dynamic operation in the accessing of databases that has not yet been achieved on a consistent basis.

It is another object of the present invention to provide a system for dynamically mapping data to objects for software applications.

It is a further object of the present invention to provide a system for mapping to an object for a software application during the run time of that application.

It is an additional object of the present invention to provide easy translations between databases and applications having a variety of different formats or data store models.

It yet another object of the present invention to provide a mapping system wherein multiple objects can be used for a particular map.

It is still a further object of the present invention to provide a mapping system wherein multiple maps can be developed for a single object.

It is again another object of the present invention to provide a mapping system wherein different data attached to an object can be selected by means of different maps.

Another object of the present invention is to provide a mapping system wherein object programming applications are tailored to delegate accessing a database and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanisms in the application code. It is a preferred object of the invention to provide such a system with a concurrent parameter setting mechanism where the runtime library repository can be set to access a particular database and to generate database specific database calls and SQL strings.

In one preferred object of the invention, a software program would be provided that can generate application programming code from database maps to provide a programming application that will delegate to a runtime library repository the functions of accessing a database and the generation of SQL strings that are specific to a database or JDBC driver.

In another preferred object of the invention, the runtime library repository can be modified and tailored to optimize database access calls and SQL strings for a particular database.

It still an additional object of the present invention to provide a mapping system wherein different maps for particular objects can be used to provide varying levels of security.

It is again another object of the present invention to provide a mapping system wherein data changes related to a particular object can be promulgated with global changes for that object, if desired.

It is still a further object of the present invention to provide a mapping system when a data map for an object can be easily edited without extensive knowledge of the relational database as a source of the data.

It is again another object of the present invention to provide a mapping system wherein the metadata describing a map of a datastore can be dynamically evaluated.

It is again another object of the present invention to provide a mapping system, wherein data can be accessed more quickly than is possible with a conventional data store accessing arrangements.

It is again a further object of the present invention to provide a mapping system in which frequently-used data can be more easily accessed than other types of data.

It is yet a further object of the present invention to provide a mapping system in which a wide variety of different data languages can be easily used.

It is still a further object of the present invention to provide a mapping system wherein virtually any type of datastore architecture can be translated so as to be useful by an object software application, or other types of software applications.

It is again a further object of the present invention to provide a mapping system wherein datastore to datastore mapping is easily facilitated.

It is still another object of the present invention to provide a fully synchronized caching system that utilizes the transaction coordination facilities of a server such as a J2EE application server (for example, Weblogic 6.1), such system comprising a local or distributed computer system having a first data source referred to as the primary data source, a second data source referred to as the cache data source and the cache data source is associated with an object to relational mapping layer to provide a data source cache for object applications, and a server having a transaction coordinator with the ability to register multiple data sources, wherein:

(a) both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and

(b) the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and

(c) registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

In a particularly preferred object of the invention the cache data source can be set up as a memory-resident entire database, a disk resident database, or both a memory resident database and disk resident which are synchronized. In one object, of the invention the memory resident database may be a portion of the disk resident database and the size of the memory for the memory resident database can be set by the user.

Another object of the present invention is to provide a database access repository, or a collection of repositories, comprising a runtime library repository having the ability to make database specific calls and to generate database specific SQL strings, and comprising other libraries, information and logic that permit mapping of information related to one or more of a member selected from the group consisting of relational databases, object databases and object programming schemas, and the like. The database access repository may contain one or more items selected from the group consisting of maps of database schemas, object definitions, other programming definitions or parameters, object metadata, database settings, complex relationship definitions. For example, metadata can be placed in a separate structure from the java object application and can be independently stored and accessed in a database access repository file collection, or in a separate file. In one preferred object of the invention, an object to relational repository can include a runtime library repository that can be optimized for a particular database and has the ability to access multiple database types, wherein the runtime library repository can be access by object programming applications that delegate database access and SQL string generation to the runtime library repository.

These and other objects and goals of the present invention are achieved by the present invention as described below, and depicted in FIG. **7**.

In some objects of the present invention, concepts are based in part upon concepts present in U.S. Pat. No. 5,857,197, (incorporated herein by reference) or that are reasonably inferable from reviewing that patent, in order for the present invention to provide an improved mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about java objects including complex java objects that have relationships with other java objects) does not need to be stored in a java object. In a preferred aspect, it is an object of the present invention to provide a database access system that does permits a java object application to delegate database accesses to a runtime library repository.

In a preferred object of the present invention, a software programming module (or modules) can automatically generate object source code from at least one database schema map, at least one object programming application schema, or from a combination of at least one database schema map and at least one object programming application schema. The code generated for the application can be set to delegate database access and SQL string generation to the runtime library repository instead of including within the application the more limiting code for non-delegated database access by the application. This arrangement allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. The mapping information can be used to map from objects to relational models or vice versa and generate appropriate code.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** and its description is a replication of FIG. **1** as shown and described in U.S. Pat. No. 5,857,197.

FIG. **2** is a concise flow chart depicting the operation of the system in U.S. Pat. No. 5,857,197 in a system comprising (a) an object language application program, (b) a data source manager and abstraction layer system embodiment of the U.S. Pat. No. 5,857,197 patented invention such as the CocoBase Enterprise O/R Software Tools Suite Version 4.0 (soon to be released) of Thought, Inc, San Francisco, Calif., and (c) a relational data source and JDBC driver. The operation of the system of FIG. **2** in the context of the flow chart operation will be described in detail below by referring to the FIG. **1** numbering system.

FIG. **3** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **2**, further including a number of system components or connection points for additional components. The wavy cloud that surrounds some of the system depicts the CocoBase components and the portion of the system that is called the Java environment. The additional component that are shown in FIG. **3** beyond the components described in FIG. **2** are (1) a CocoBase Object Access and Storage Adapter software module, (2) a CocoBase Object Model Repository (may be stored in a separate data source or in the system DataSource Relational Database), (3) a Plugin API (and optional caching database attached via the API) to the CocoBase Mapping Layer for caching one or more members of the group consisting of database data, query string(s), CocoBase Map(s), and data object(s), and (4) a CocoBase Dynamic Object/Relational Respository as a separate file, which could also be stored in the DataSource Relational DataBase.

FIG. **4** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **3**, further including a number of system components or connection points for additional components. The additional compo-

nents shown are (1) an CocoBase Development Software module called CocoAdmin having command line and/or GUI interfaces for accessing, editing and creating new maps, for generating source code, for compiling object classes from source code, or for generating J2EE components such as EBJs from CocoBase maps, (2) one or both of a CocoBase CocoDriver Server, (3) Object Access and Storage Adapter for processing the storage of objects or object schema(s), which may access or process them in the form of objects for an object database, as a UML file for a UML modeling tool, or may convert the objects or object scheme to or from one or more of UML, XML or XMI format representations of such objects or object schema(s), (3) an optional CocoBase Object Model Repository file, which may be stored as stand alone files, in an Object or Relational data source, or an XML repository, and (4) an optional Object Modeling Tool Data File which can be exchanged between the CocoBase Tool Suite and the Object Modeling Tool in one or more directions, i.e., to the CocoBase tool suite, from the CocoBase tool suite, or both to and from the CocoBase tool suite.

FIG. **5** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **4**, shown without the optional Object Modeling Tool Data File component, but further including a number of system components or connection points for additional components. The additional components are useful in a caching system or in a system for porting data from a first data source to a second data source. These additional components shown in FIG. **5** are (1) a second JDBC driver connection, and (2) a Second DataSource or Cache DataBase, where the Cache database may exist completely or in part in the system RAM, and may concurrently have a fixed storage medium component for the caching system.

FIG. **6** is a diagram depicting the comprehensive operation of the present invention that combines all of the components described in FIGS. **2**–**5**, or lists them as options within the diagram. This diagram shows a system having both a DataSource Relational DataBase and an Optional Object Database DataSource, as well as an Optional CocoBase Data Migration Tool. This migration tool can migrate and translate data between each of an Object database, a Relational database, an XML storage file, and a flat text file XMI file. An optional migration tool has the ability to migrate and translate to and from a UML format and an XMI format. Other system components have been described further in FIGS. **2**–**5**. Also, how to implement or remove various components in a system as shown in FIG. **6** is further described below in the detailed description of the invention.

FIG. **7** is a concise flow chart diagram depicting the comprehensive operation of the present invention, and including a number of variations on the basic system described in FIG. **2**. The components in FIG. **7** that have the same numbering as components in FIG. **2** will have the same description as those components as described for FIG. **2**. In general, the invention of FIG. **7**, includes (a) a new component or set of components and/or logical steps (**3**A) for caching objects, data or maps, (b) a new component or set of components and/or logical steps (**7**, **51**, **52**, and **8**) for evaluating XML information and for extracting objects, data or maps from the XML information for placement in a object (information can be in XML format) and sending the object to the Object Application **101** in response to request (**2**A), (c) an new component or set of components and/or logical steps (**4**B) for converting objects, object schema, data, meta data, or maps into one or more of an XML file or into an XMI file.

DETAILED DESCRIPTION OF THE DRAWINGS

The following detailed description may utilized terms that are defined in the "Definitions" section which immediately follows this section.

Detailed Description of FIG. 1

FIG. **1** illustrates one such embodiment involving a request **100** by an object application **101** and the processing of such request. The system according to one embodiment of the present invention comprises an object schema **200** including meta data **201** corresponding to a data store schema **300**, a first adapter **400**, and a second adapter **500**. In the preferred embodiment of the subject invention the object application **101** is a java applet "cocodemo java" as listed in Appendix A of U.S. Pat. No. 5,857,197 ('197 patent), issued Jan. 5, 1999.

One embodiment of the '197 patent invention used a user defined object view of an underlying non-object store, i.e. the object schema **200**. The object schema **200** was accessed by an object application **101** through an abstract layer **600** which does the necessary conversions between object and non-object views. This abstraction allowed both kinds of data stores—object and non-object (e.g., relational)—to be accessed identically from at least one object application **101**.

Moreover, by applying simple object streaming, the '197 patent invention was capable of accessing objects distributed over a three tier environment using exactly the same application program interface ("API ") **700** as the local two-tier environment. This wa accomplished by using access "adaptive" technology comprising the adapter abstraction layer **600**.

The adapter abstraction layer **600** of the '197 patent performs any translation work necessary for converting objects to both object data stores **312** and non-object data stores **302**. The adapter abstraction layer **600** provided a consistent API **700** for both object and non-object data stores and enables application programmers to migrate between various object stores without application modification. The adapter abstraction layer **600** also facilitated communication with a remotely available second adapter **500** without modifying the object application **101** programming logic.

In one embodiment, the '197 patent includes the use of meta data **201** (i.e., data used to describe other data) to define how to access and convert non-object data store content **304** to objects and back. This was accomplished by paring down the non-object data store schema **300** into its various components including tables, fields, and conditions **305** in one embodiment. The paring made the creation, management, and access to the meta data **201** to be a convenient and elegant task. By using the adapter abstraction layer **600** which understands and uses the meta data **201**, the '197 patent provided an abstract view of an underlying data store(s) **302** (**312**, **322**) as an object store. The effected abstraction produced an architecture whereby the underlying data store(s) **302** (**312**) as well as multi-tier adapters (e.g., **400**, **500**, **4**XX, and **5**XX) could be interchanged without object application **101** code modification.

One embodiment of the '197 patent was implemented using the adapter technology and was therefore capable of being deployed as either a 2-tier or as a 3-tier environment transparently. The adapter abstraction layer **600**, and more specifically the first adapter **400**, communicated with a server process, more specifically the second adapter **500**, in communication with the underlying data store **302**, which transferred the requested data store content **304** to and from the first adapter **400** in communication with object application **101** client. The first adapter **400** instantiated the object **112** from more primitive data types comprising data **115**, reflecting the data store content **304** obtained from the server process, more specifically the second adapter **500**. The method of breaking down objects **102** (**112**) into the corre-

sponding primitive types comprising data **105** (**115**) ensured successful transfers of any kind of object irrespective of object application **101** views of the object(s)' data elements. The '197 patent invention used the adapter technology to dynamically load data store access code **205** at runtime. This architecture requires objects to serialize themselves in order to be persistable or for the computer language and language runtime to be able to inspect and set an object's structure in such a way as to be able to automatically persist the object's contents. Persistable is a term of art which means an object's contents can be saved to some more permanent location such as a computer disk, and in this case refers to the storage and retrieval of the object's contents from a data store. As a result, the present invention can readily adapt to an N-tier adapter solution which accesses the data store through some intermediate data server, thereby increasing deployment and scalability options.

FIG. 1 of the '197 patent shows one embodiment of the '197 patent invention which worked in conjunction with an object schema manager **203**. The object schema manager **203** allowed data store administrators to create and maintain an object view of data store content **301** stored in the data store **302**. The object schema manager **203** could read, write, and update the meta data **201**. One embodiment of the '197 patent invention was directed to accessing non-object (e.g., relational) data stores **302**, creating meta data **201** based on data store schema **300** comprising tables, fields and clauses or conditions **305**, making up the data store **302**. One such embodiment of the '197 patent comprised a set of tools which were designed to create and maintain the meta data **201** by storing the information in a repository (such as the data store **302** itself), which information was then loaded by the second adapter at runtime. This meta data **201** then served as a map for operations available on the data store(s) **302** (**312**), for the way these operations should be performed and under what conditions.

FIG. 1 of the '197 patent further illustrated how the object schema manager **203** facilitates the creation and maintenance of the object schema **200**, thereby increasing productivity and reducing the opportunities for typographical errors, which can create preventable runtime errors. The object schema manager **203** permitted dynamic modification of object schema **200** without requiring modification or recompile of the object application **101** to ensure that the object application clients were not 'brittle' or limited to a particular database or object model as is traditionally the case in data store access applications. One embodiment of the '197 patent used the meta data **201** to construct an object view of the data store schema **300**, e.g., an object schema **200**, and used the meta data **201** to generate one or more commands **303** for accessing the data store content **304**. The deployment of meta data **201** in this way in one embodiment had the effect of using the non-object data store(s) **302** to be described and accessed as object(s) **102**, though its contents might have been more rudimentary and abstract.

One embodiment of the '197 patent stored meta data **201** in a data store **204** and dynamically loaded and executed meta data **201** at runtime. As a result, in one embodiment of the '197 patent, access code **205** corresponding to the command(s) **303** is centrally created and managed in a central repository, i.e., data store **204**. Such central repository architecture allowed the object schema(s) **200** to be re-used across objects **102** (**112**) and object applications **101** (**111**, **121**). Such an approach also allowed for any changes of one object schema **200** to be transparently reflected to any client object application **101** accessing and using the object schema **200**, without the need for object application **101** modification or recompilation. The '197 patent contemplated a notification mechanism so that data store **302** modifications resulting from requests in one location could be immediately communicated to all object applications **101** accessing that object schema **200**.

One embodiment of the '197 patent facilitated the separation of data store access code **205** (such as Structured Query Language "SQL") from the object application **101** logic. This facilitated simple and rapid application development and lowered maintenance cost for a computer system that required database access by improving the flexibility of the object application **101** and by reducing the brittleness of the object application **101** client(s). As a result, users of the invention described in the '197 patent could attain a cost savings by centralizing the task, reducing the possibility of typographical errors, installation problems, and skills availability shortages. The object stores could be created and managed by someone with data store (i.e., relational) expertise (as most modem organizations have) without requiring those individuals to also have object programming expertise.

One embodiment of the '197 patent resulted in an architecture for centrally storing and managing object schema **200** to promote re-use, effect enhanced performance, and better use of (S) engineering and administrative personnel skills. By way of example, storing the access code **205** within the physical confines of the non-object data store **302** in one embodiment, created a centrally manageable and maintainable system for accessing that non-object store **302**. The object schema manager **203** thus allowed for better use of the MIS personnel's skill sets by focusing on data access in terms familiar to people maintaining their respective system. That is, one embodiment of the '197 patent allowed object application engineers to concentrate on applications code without concern for or need to learn SQL or any other native access code **205** of the data store **302**. At the same time, data store administrators could concentrate on managing and maintaining the data store(s) **302** without knowledge of application programming and without concern that any changes in the data store content **301** or data store schema **300** would adversely affect the object application(s) **101** in communication with the data store(s) **302** according to one embodiment of the '197 patent.

A central management architecture provided by one embodiment of the '197 patent allowed data store specialists to see, manage and control all access to the data store(s) **302** down to the individual access code **205** commands. It also allowed for the fine tuning and reconfiguration of those commands by the data store specialists, without involvement of object application developers. The architecture allowed the best skills to complete the respective task at hand without impacting or being impacted by the lack of available skills that an individual may not have in the other area.

In one embodiment of the '197 patent, the object schema manager **203** allowed for the arbitrary creation and management of object schema(s) **200** via a point and click interface, thereby significantly increasing the efficiency of creating the data store access code **205**, and improving the reliability of the overall system. This embodiment also allowed completely arbitrary joining and combining of the data store content **301** as a single object view of that data, and overcame the constrained and often uselessly arbitrary or non-existent foreign key constraints of conventional mapping systems and data stores. These hard-coded links such as foreign key constraints may not be supported properly by all underlying data stores, and not be implemented by the specific customer's data store (often legacy) implementation. New foreign key constraints are often not

possible on some data stores because they may not work properly with older applications if the constraints were to be created and enforced.

As a point of reference, a foreign key is generally accepted as a predefined relationship or link which the data store manages in order to ensure that records are properly related in a database. A simple example would be to have a foreign key between an order number in an order table, and an order number in a line item table which keeps the individual items belonging to that order. The foreign key in this example would be between the order numbers in both tables, which might not allow, for example, a line item record to be entered if there is no order in the order table corresponding to the order number.

A join using this embodiment of the '197 patent may occur on any arbitrary field of any table within the data store schema **300**, even if a foreign key does not exist between the fields of the two tables to be joined. Such flexibility is immeasurably useful when combined with the ability of the object schema manager **203** to access more than one data store **302** at the same time. The resulting architecture of the '197 patent allowed data store content **301** from one data store **302** to be joined with data store content **311** from another data store **312** through the object schema manager **203** without requiring any data store schema **300** changes to either system whose content is being joined. Such transparent and independent flexibility is of utmost value for the many diverse data stores currently operated by most large and small business and other entities.

One embodiment of the '197 patent provided a way to manage the data store access code **205**, requiring only an understanding of the data store **302** and the data store schema **300**, instead of requiring object programming knowledge within the modem working environment.

Once the object schema **200** was created, the object application **101** running in the application environment **103** sends a request **100** for an object **102** (possibly including an object data with lookup attributes to be matched) located in at least one data store such as an object data store **312** or a relational data store **302** in one embodiment of the subject invention. At a very high level, the request **100** was dynamically translated to a data store specific request which was then executed and results of which were parsed and new data object(s) **112** created and initialized with the results of the executed request **100**.

In one embodiment of the '197 patent, the request **100** and the accompanying object '197 patent was passed from the object application **101** to the adapter abstraction layer **600** comprising the first adapter **400**. The first adapter **400** then extracted the object attributes **103** and the object name **104** from the object **102**, and packed the object attributes **103** and the object name **104** as data **105** to be used in communication and transport layers. The first adapter **400** then communicated the data **105** and the request **100** to the second adapter **500**. The communication medium **630** between the first adapter **400** and the second adapter **500** may comprise an Internet connection, whereby the second adapter **500** is operating inside a firewall. Logical division **650** comprised a logical separation between a client, e.g., the object application **101**, and a server, e.g., the abstraction layer **600** and more specifically the second adapter **500**.

The second adapter **500** searched a meta data map **206** comprising at least one object name **114** using the object name **104** to determine whether the object name **104** existed in the meta data map **206**. If the meta data map **206** contained the object name **104**, then the second adapter **500** used the object attributes **103** and the found meta data map **201** to generate at least one command **303** for accessing the data store **302** according to the request **100**, if the command has been enabled, ensuring further security by restricting access. By way of example, the subject invention contemplated the use of this technology to provide read only data stores over the Internet, irrespective of the user's standard privileges within the context of the data store.

The second adapter **500** then executed at least one such command **303** as a data store access client server process **700** in communication with the data store **302** using a JDBC driver to which is passed data store access code **205** and obtains the data store content **301** and an execution status **306** based on executing at least one such command **303**.

The second adapter **500** then processed the data store content **304** and the execution status **306** using meta data **201**, and packed the obtained data store content **304** and the execution status **306** as data **115**. The second adapter **500** communicated the data **115** to the first adapter **400**.

The first adapter **400** unpacked the data **115** and instantiated the object attributes **113** and the object name **114** into potentially one or more new object(s) **112** and/or status according to the request **100** and the data **115**. The first adapter **400** then communicated the request **100** and at least one such new object **112** and/or status, if object results were found, from the first adapter **400** to the application program **101**.

Several points should be noted regarding the implementation of the '197 patent invention in the preferred embodiment. As each object(s) **102** data was being read from the data store(s) **302** (**312**), an instance of the object's class type (or potentially some generic container object) was instantiated (in the preferred embodiment a Java Class is instantiated for each set row result returned) and initialized to the row of attributes returned in the result set. Because Java can instantiate new objects based on class name, objects could be managed by a preferred embodiment of '197 patent even though it may have been unaware of their existence or composition at compile time. For ease of implementation, each java class in order to be supported by the preferred embodiment must implement a standard interface. With compliance to this standard interface, the subject invention can call the methods necessary to initialize the data automatically, making new instances automatic and transparent to the client object application **101**.

Implementation of explicit reference handling was not required using the subject invention, and therefore is not implemented in the preferred embodiment. Unlike more rigid systems which limit the type of operation and mechanisms which will and can be used to access data stores, the subject invention allows for flexibility. This is to reflect the market reality that most organizations operate large systems in place, which should be capable of being accessed from a variety of sources in a variety of different ways. Any attempts to manage references in these mixed environments would not only prove a futile exercise, but would complicate and reduce reliability, without providing the desired functionality. As for reference handling within the object application, the Java language, which was used to implement the preferred embodiment, has language supported reference counting within its memory management and garbage collection software. It is contemplated by the subject invention, however, that specific adapters and data objects may be implemented with this architecture for specific data stores supporting appropriate reference handling management.

A preferred embodiment of the '197 Patent used an in-memory list to maintain object references, which list could be either partially or fully pre-fetched. An object application may retrieve one object, all of the objects at once, or in blocks of default, or in blocks of a user defined size. Once the objects were in the memory list, they could be iterated through using their offset in the list as their index. The '197 patent contemplated incorporating smart vectors, which would automatically fetch Objects as they are requested from the vector itself, instead of the object application manually having to go back to the data store to request new blocks.

A preferred embodiment of the '197 patent provided non-granular simple transaction support, although the '197 patent invention contemplated either building its own transaction management system, or using a pass through mechanism which allowed for more sophisticated transaction management to be written by the application programmer, if the underlying system supports it. Any limitations due to the preferred embodiment of the '197 patent being implemented in the Java Driver Database Layer and JDBC or the Java DataBase Connectivity Layer were not contemplated by the invention in the '197 patent. Thus, the '197 patent contemplatee using more advanced transaction management, as available in the implementing language and architecture.

When the '197 patent invention was used to access the non-object data store **302**, e.g., relational databases, the object schema **200** providing meta data **201** descriptions of how to map to the underlying data store **302** was used to generate the native access code **205** necessary to access the data store content **304**. One embodiment of the '197 patent used the object schema **200** to build an SQL statement at runtime, which was then executed to get the desired results, based on the request **100**. Because multiple statements are likely to be executed against a given class type after it has been first accessed, the '197 patent can cache the meta data for the class (name) in memory, within the second adapter **500** and saves pre-compiled versions of the access code **205** which can simply be re-executed without recompile. This approach provided improved performance characteristics for successive accesses. In the three-tier design embodiment of the subject invention, the cache was physically located on the server, and was therefore shared across all client accesses to that server, providing a performance optimization in an area where performance is most critical.

The '197 patent invention comprised the adapter abstraction layer **600** comprising a set of runtime adapters (e.g., the first adapter **400**, the second adapter **500**, the n-th adapter **4**XX, **5**XX), which can be transparently interchanged by the object application **101**. The adapters serve various functions such as direct 2-tier data access, or 3-tier or even n-tier gateway/transport. Because all adapters implement the same API **700**, they are interchangeable in the object application **101**, providing new and useful functionality, beyond implementations described in the '197 patent, without requiring object application **101** modification. Object application **101** runtime flags can instantiate entirely new adapters which might interface with radically different data store(s) **302** without modification to the object application **101**.

When data store access code, e.g. SQL, is dynamic SQL instead of pre-compiled SQL, there are typically some performance penalties to be paid for dynamic execution. One preferred embodiment of the '197 patent addresses these performance issues by pre-compiling SQL the first time a given SQL operation was executed, and by preserving the compiled statement within the second adapter **500** for future executions from the same or a different first adapter **400**. The '197 patent invention binds variables to the statement, and re-issues the generated access code with the new SQL. As a result, the penalty as to dynamic execution became a slight overhead the first time an object **102** of a given type was requested from the abstraction layer **600**. Past this first point the performance characteristics should be the same as from any other embedded SQL client, perhaps even better because the amount of information to be processed and transferred to the server accessing the data store **302** was likely to be less using the architecture of the '197 patent than one which embeds SQL within a client object application and ships it with every request, and then must process the results inline, and often unnecessary descriptive data that is returned, and which is unnecessary in a system which is already preprogrammed with and understanding of the composition of the return values.

Because the first adapter **300** packed object request(s) **100** as a single API **700** call returning a single complex result (building the resulting data objects **112** transparently in the client from the returned complex data **115** packet), the '197 patent invention reduces network latency (which is essential to reduce over the Internet and other io-bound and high latency networks) and increases the performance and scalability of the involved clients and servers. The '197 patent invention also reduces the quantity of extraneous information which is always returned to client object applications by other prior art systems such as the JDBC type-3 distributed database access layer architecture upon which other common lava Database Products are currently built. As a result, not only did the '197 patent architecture obtain tremendous performance gains by reducing the latency of multiple network roundtrips, it also reduced the quantity of data and therefore increased the number of clients it could support, since each request/response required less I/O. Compression and data duplication reduction techniques were contemplated by in the '197 patent to further increase performance.

New adapters, e.g., **4**XX, **5**XX, could be added to the adapter abstraction layer **600** dynamically, even at runtime, as long as they complied with the API **700**. This allowed data store(s) **302** (**312**) to have custom adapters built for them (such as object data stores or other relational and/or non-relational data sources) which comply with the API **700** of the subject invention and are loadable in the same way as the adapter abstraction layer **600** described in the '197 patent.

Detailed Description of FIG. **2**

FIG. **2** is a concise flow chart depicting the operation of a computer system having an implementation of the "System and Method for Accessing Data Stores as Objects" described in U.S. Pat. No. 5,857,197. The system of FIG. **2** comprises three basic components (a) an object language application program, (b) a data source manager and abstraction layer system embodiment of the U.S. Pat. No. 5,857,197 patented invention (such as the soon to be released CocoBase Enterprise O/R Software Tools Suite Version 4.0 of Thought, Inc, San Francisco, Calif.), (c) a relational data source and corresponding JDBC driver, and (d) an optional Maps Files Storage facility. The operation of the system of FIG. **2** in the context of the flow chart operation will be described in detail below by referring to the FIG. **1** numbering system that was described above.

FIG. **2** starts at position IA (Start/End) with a request step **2**A from an object application (**101** in FIG. **1**), which calls the CocoBase DataSource Manager and Abstraction layer (hereafter CocoBase system) and delegates the database accessing to CocoBase. The logical loop that the CocoBase system performs begins in response to the request step (**2**A) and first performs the Extract Object Name step (**4**). The

CocoBase system loop continues onward to the Evaluate Object Properties step (**5**) and its associated components where (i) object properties are evaluated (**5**), (ii) object to relational maps are accessed, created, or updated by **3**B (and this may involve retrieving maps from the system DataStore **3**C or from a separate Optional Maps Files Storage facility (**1**C)), and (iii) database commands necessary for data retrieval (including SQL strings) are generated at step **2**B under the control of the Execution Status Monitor (**1**B), before the CocoBase system continues onward to the Obtain Data step (**6**). In step **6**, the CocoBase system in coordination with components **1**B, **2**B and **3**B, passes SQL strings or other DataBase Access Statements from **2**B to the JDBC Driver (**2**C), which obtains data from the system DataStore (**3**C) and passes that data to the CocoBase system. Then the CocoBase system places the obtained data in an object (Step **6**A, Place Data in Object and sends the Data Object (Step **7**A, Send Data Object) to the Requestor Object Application (back to Step **2**A position) and the delegation to the CocoBase system by the Object Application ends (back to Step **1**A).

In the paragraph above, the general functions of the system of FIG. **2** have been explained. In the paragraphs below are more detailed descriptions for the (un-illustrated) CocoBase system components that may be involved in each step of the flow chart (or CocoBase components associated with that step as illustrated in FIG. **2**) while using FIG. **1** reference numerals to better describe parts and functions of those CocoBase system components.

At step **1**A, the Object Application (**101**) may contain two referenced parts (a) Object Schema (**200**) and (b) Meta Data (**201**), and its Request (**100**) for data at Step **2**A, which is delegated to the CocoBase system.

At Steps **4** and **5**, the Abstraction Layer Adapter **1** (**400**) may involve the Object Schema Manager (**203**), Object Schema (**200**), object data store contents (**301**) of DataStore (**302**) or in a separate external mapping repository storage facility (e.g., Optional Maps Files Storage (**1**C)) to obtain the Object Name (**104**) and Object Attributes (**103**) for objects (**102**) or (**112**) as data (**105**) to be used by the CocoBase system. Particularly at Step **5**, the data (**105**) may be passed (optionally across a logical division **650**) to a second Abstraction Layer adapter (**500**) to an abstraction layer portion **600**. The second Abstraction Layer adapter (**500**) checks the meta data maps (**201** or **206**), which may be an Object/Relational map. The adapter (**500**) uses the object attributes (**103**) and the data map (**201** or **206**) to generate at least one command (**303**) such as an SQL string or other DataBase Access Statements (**2**B) under the control of the Execution Status Monitor (**1**B) of the adapter **500** for accessing the database such as the relational DataStore (**302**/**3**C).

The second adapter **500**, at Step **6** of FIG. **2**, then executes at least one command **303** as part of a data store access client server process **700** in communication with a JDBC driver (item **2**C in FIG. **2**) which converts the at least one command into a format that is understood by the data store **302** (item **3**C in FIG. **2**). In **2**C of FIG. **2** the JDBC driver communicates with the data store item **3**C by using data store access code **205** and obtains at least one item of data store content **301** and an execution status **306** based on executing at least one such command **303**. The client server process **700** passes this data store content **301** and execution status **306** back to item **6** (Obtain Data) of the CocoBase system which coordinates with both the client server process **700** of CocoBase and the Execution Status Monitor component of FIG. **2** (item **1**B) to provide this information to the second adaptor **500** of the CocoBase system.

The second adapter **500** then processes both the obtained data store content **304** (collection of at least one item of data store content **301**) that corresponds to the request and the execution status **306** by using meta data **201** (of **3**B in FIG. **2**), and packs the obtained data store content **304** and the execution status **306** as data **115**. The second adapter **500** of Step **6** in FIG. **2** communicates the data **115** to the first adapter **400** of Step **6**A in FIG. **2**.

In Step **6**A of FIG. **2**, the first adapter **400** unpacks the data **115** of Step **6**, and instantiates a set of object attributes **113** and the object name **114** (object attributes and object name are obtained in Steps **4**–**6** of FIG. **2**, see description above) into potentially one or more new object(s) **112** (see FIG. **2**, Step **6**A "Place Data in Object) and/or status according to the request **100** (the original request from Object Application **101** for data from Step **2**A of FIG. **2**) and the data **115**.

In Step **7**A of FIG. **2**, the first adapter **400** of the CocoBase system then communicates the request **100** and at least one such new object **112** and/or status, if object results were found, from the first adapter **400** to the application program **101** request of Step **2**A in FIG. **2**. The Object Application **101** at **2**A indicates receipt of the information to the CocoBase system and at Step **1**A the Object Application completes and ends the data request process.

Detailed Description of FIGS. **3**–**6**

FIGS. **3**–**6** illustrate more complex variations of the CocoBase system that is described in detail above with respect to FIG. **2**. Different components and features are added to the basic system of FIG. **2** to provide features and functions that are described below in detail in the detailed description of the invention. See the description below for instructions on how to add or remove such additional CocoBase system components and their features to individual computer systems. Such details are provided below contextually for the individual components.

Detailed Description of FIG. **7**

As indicated above, FIG. **7** is a concise flow chart diagram depicting the comprehensive operation of the present invention, and including a number of variations on the basic system described in FIG. **2**. The components in FIG. **7** that have the same numbering as components in FIG. **2** will have the same description as those components as described for FIG. **2**. In general, the invention of FIG. **7**, includes (a) a new component or set of components and/or logical steps (**3**A) for caching objects, data or maps, (b) a new component or set of components and/or logical steps (**7**, **51**, **52**, and **8**) for evaluating XML information and for extracting objects, data or maps from the XML information for placement in a object (information can be in XML format) and sending the object to the Object Application **101** in response to request (**2**A), (c) an new component or set of components and/or logical steps (**4**B) for converting objects, object schema, data, meta data, or maps into one or more of an XML file or into an XMI file.

These additional components and features of FIG. **7** that are added beyond the basic system of FIG. **2** relate to providing features and functions that are described more fully below in the detailed description of the invention. See the description below for instructions on how to add or remove such additional CocoBase system components and their features to individual computer systems. Such details are provided below contextually as relates to the individual components of FIG. **7**.

DEFINITIONS

For the purposes of the present application, the following definitions are given as a meaning for terms used herein

throughout this application to avoid any confusion with possible multiple meanings for such terms. Other terms used herein have meanings that are well recognized in the art, and their meanings will be clear from the context in which they are used in this application.

A "module" in the computer programming context is an organized set of computer code designed to act on some externally passed in data, where everything needed for that action is passed in to the module.

An "object" in the object oriented programming context is an organized set of encapsulated programming code designed to act on itself at the request of some external system, which system may pass in some additional information to the object when it delegates a task to the object in that request.

A "composite object", or an object programming "component", in the object programming context each refer to an object comprising a complex and organized set of objects that are encapsulated to form the composite object. The two terms "composite object" and "component" (e.g., a J2EE component such as an EJB, Enterprise Java Bean) may be utilized in an interchangeable manner in referring to the same type of logical constructs and concepts.

A "delegation" in the object oriented programming context is where an object or a programming application permits another simple object, set of objects, composite object, or a module to perform an action by simply requesting the action from the delegated simple object, set of objects, composite object or a module.

A "non-delegation" database access in the object oriented programming context is where an object or a programming application has specific code imbedded in the application which directly controls database calls and the generation of SQL strings, wherein the imbedded code is specifically tailored for accessing a particular database or for a specific database schema.

A "user interface" for an object oriented application, such as a Java Server Page (JSP), a Swing GUI, and the like, refers to a software component or module that provides a feature for a user that will permit the user to interact with an object or programming application in some way, such as the interactions of finding, selecting, inserting, updating and deleting data in a database.

A "library" is a set of definitions, data, objects or programming modules that may be accessed by a computer programming application to obtain information or to delegate tasks.

A "repository" in the object programming context and in the context of this application is a special set of libraries that may include, among other things, items related to object to object mapping, object to relational mapping and database accessing information, database parameters, optimized database access routines, and the like. A repository may be a single file or may be a set of files. The format of items in a repository may vary widely according to the desire of a computer programmer user or developer and may be in one or more of formats such as simple text, XML, XMI, UML, JDBC, source code, compiled code, and the like.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is based in part on U.S. Pat. No. 5,857,197, (incorporated herein by reference), and provides a mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about java objects including complex java objects that have relationships with other java objects) does not need to be stored in a java object. Instead, the mapping information related to object definitions and some metadata can be placed in a separate structure that is independently stored. This allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. The mapping information can be used to map from objects to relational models or vice versa, objects to objects, object to COBAL or vice versa, and object to XML and the like.

Certain of the embodiments of the present invention are embodied in a suite of products by Thought, Inc., referred to as "CocoBase". Aspects of the CocoBase technology are also described in U.S. Pat. No. 5,857,197, supra. Also, a number of details of the system of the present invention are provided now as state of the art in the documents that have been posted on the website of Thought, Inc. or are otherwise available on the internet as publications.

In one embodiment, the mapping information, rules, or metadata can be maintained in a human-readable format and can be interpreted by an application to apply the rules to data objects. This information can be stored directly in the data source or can be stored in a separate file. In either case, this information is called a "repository". This repository provides an easily maintainable and transportable format for the mapping rules. The rules and java object relationships can be updated, or otherwise changed, dynamically even while other rules or object relationships in the repository are being used to perform conversions. The rules and relationship definitions can be embedded in standard formats such as in Java code, e-mailed, sent as plain text, etc. This allows for flexible transfer and proliferation of the rules to other sites where the rules can be used to access objects (which can be bundled with the rules), used to derive other rules, etc. In a particularly preferred embodiment, this repository is in an XML (extensible markup language) format, and the object model definitions can also be present in an XMI (XML metadata interchange) format or can be easily exported to an XMI file format.

A more preferred embodiment is such a system and software as provided by an updated CocoBase Enterprise Object/Relation Software package (hereafter CocoBase), which includes the software tool CocoAdmine. Such updated software package, is available, or is available shortly, from Thought, Inc., San Francisco, Calif. An embodiment of the invention using this package is described below.

Exporting Maps as XML Repositories in CocoBase

CocoAdmin provides a mechanism for the export of maps defined against a database into a modifiable XML format. This facility allows a system user to select an existing map(s) to be exported, and to also specify the filename of the XML document to be created. The resulting XML document is written to a file with the specified filename using an XML template named coco.dtd that is ordinarily located in the thought\cocodemo3tier31\demos\resources directory.

Once the XML file is written, it can be edited using a standard text editor or XML editor, and can be modified to reflect map customization requirements. The following discussion relates to specific features of the generated XML file.

The basic repository format has CBObject (CocoBase object class) definitions that reflect the select, insert, update,

delete and call related map definitions for a CocoBase map. Each of those operations further consists of tables, fields and clauses that may exist to specify how the object is mapped to and from the data source.

XML files contain a DTD (document type definition) entry which describes the structure of the tags and data contained in the file. The DTD only checks for the existence of required elements, but doesn't check the contents of those elements. Given this desired degree of freedom, it is possible to enter invalid map information which will not be processed properly by the CocoBase runtime and care should be exercised to avoid improper editing. In general, modifications should be restricted to the schema, table, and field name variables in the Tables and Fields entries. These variables may require different values when they are exported from one database instance, and imported into another.

To begin exporting a set of maps, select File-> Export XML Map Repository from the CocoAdmin pull-down menu. Multiple maps can be selected by holding down either the <Shift> or <Control> when selecting the connection and map(s) to export.

When Next > is pressed in the dialog box provided, the list of selected maps will be presented in a list and the window will prompt for a final acknowledgement of export. The XML repository filename including the directory path can be specified in this window. If the directory path is not specified, the XML repository will be written to the thought\cocodemos3tier3\demos directory. When the Export XML > button in the dialog is pressed, CocoBase creates the XML file in the specified directory. After the document has been written to the disk a dialog appears which acknowledges that the repository has been created.

Importing Maps from XML Repositories Using CocoBase

CocoAdmin provides a mechanism for importing XML based CocoBase map definitions, or for importing an XMI object model definitions and then generating corresponding XML based CocoBase map definitions. Using the XML syntax defined in the thought\cocodemo3tier31\demos\resources\coco.dtd template file, CocoAdmin can import maps previously defined and exported from a different database or from a different object instance. The XML files generated from CocoAdmin will be validated against the DTD, and a basic syntax check will occur. No in depth syntax checking will occur because of the flexibility allowed in the system. When an import of an XML map definition occurs, it is in relation to an open database connection.

To begin importing an XML map definition from the CocoAdmin GUI select File-> Import XML Map Repository from the pull-down menu or Import XML Map Repository from the popup menu in the main CocoAdmin dialog. Import an XML based map definition by selecting the database connection into which the import is to occur, and click the Browse button to find the XML repository file to be opened. After a file is selected the dialog button Load XML Document is clicked and the selected XML filename appears in the initial Import dialog. After the Next > button is pressed, the selected maps are compared with those already in the database. If a newer version of a map already exists in the database, by default a flag is set to retain the newer version. If the maps being imported are newer than the ones already in the database, then by default, a flag is set to import each of the newer ones. When the comparison operation is completed the user can override any flags before the task is initiated. For example, if the XML maps are older than the versions already in the database, this condition causes the Already Exists? flag to be checked and the XML map definitions will not be imported unless the user overrides this flag by checking the Import/Overwrite? box.

When a user clicks the Import XML Repository button as described above, maps marked for import will automatically be integrated into the list of maps in the CocoBase repository for that database. If the imported maps do not reflect the physical structure of the underlying database, they may need to be manually edited by the map editor before using them with an application. Also, XML format maps can be edited before they are imported. For example, an XML editor or any standard text editor can be used for this operation. Most common edits consist of changing column names, table names, and less frequently, the schema names of the tables and the fields that the map may access.

The CocoBase Programmer's Guide available at the www.thoughtinc.com website provides further information about how to work with XML format maps and CocoBase, which published document is incorporated herein by reference. This document also explains how to set CocoBase (or CocoAdmin of CocoBase) and its runtime modules to cache maps that an application will be using in order to speed up user access. Other user hints and instructions are provided therein. For example, one URL option in CocoBase is to specify an XML repository for the maps that an application will be using. Instead of using the database repository, the runtime can use an XML repository exclusively. This technique allows CocoBase to execute against a production database that cannot be modified with map definition tables. Optional mapping server plug-ins are also described.

Data source maps according to the present invention may be utilized to generate application programming code, such as Java source code. For example, the CocoAdmin tool will allow a user to generate Java for a range of targets including most commercial Application Servers and Enterprise Java Bean Servers. A complete list of pre-configured targets can be viewed from the Connections window in the Generate Java wizard of CocoAdmin. Any Java class can readily be used with CocoBase, whether or not it was generated by the CocoAdmin tool. However, there may be significant advantages that occur from using the code generation facilities of CocoBase to quickly generate your Java classes. For example, the disadvantages inherent with data source specific non-delegation database access code can be avoided by generating code, which delegates the database access functions to CocoBase runtime libraries. Only CocoBase is believed to utilize this type of code that results in dynamic O/R mapping system capabilities.

The CocoAdmin tool can use a combination of the map definition and database foreign key relationships (if any exist) to define how the Java class is going to be generated. The Generate Java wizard of CocoAdmin can be started in one of four ways to generate Java code from a CocoBase map.

1. Clicking the on the coffee pot icon located on the CocoAdmin tool bar.
2. Selecting File-> Generate Java Code from the CocoAdmin pull down menu.
3. Selecting Generate Java from Existing Map from the launch pad after a connection has been made to a database repository, then clicking Launch.
4. Right clicking in the CocoAdmin main window and selecting Generate Java Code from Map from the pop-up menu.

From the connections directory tree in the Generate Java wizard, select the map from which a Java class will be generated. From this window, the CocoAdmin user can also

set flags to automatically generate code that does one or more of the following:

Supports the transaction object (check TransObj).

Uses existing foreign keys to automatically determine object relationships (check ForeignKeys).

Uses the CBDrop, CBProp or both of CocoBase persistence interfaces (check CBProp and/or CBDrop).

The CocoAdmin user can also (optionally) enter in a package prefix at this point. Java classes are typically organized into packages, which are stored in a common directory. The package prefix (entered at the Pkg Prefix: prompt) will then refer to the directory in which the Java code will be stored.

The Code Generation Template drop-down list of the CocoAdmin code generation wizards allows a CocoAdmin user to select a target for the classes (Java code) that will be generated for a member of the comprehensive list of targets. The following target categories are supported for most commercially available application servers and EJB servers.

Java for distributed objects and application server applications.

Enterprise Java Beans with Bean Managed Persistence (BMP).

Enterprise Java Beans with Container Managed Persistence (CMP).

The Generic EJB Entity Bean CMP—All Parts option from the drop down of CocoAdmin will generate a completely standard CMP Bean, which can be installed using any vendor supplied CMP installer. CMP concepts for EJBs are discussed in a variety of documents published in this field, including in the appropriate section of the CocoBase Programmers Guide. In order to take full advantage of the CocoBase OAR mapping features, such as delegated database access and the like, a user must install the generic CMP with the CocoBase CMP installer tool, which is an optional component of the CocoBase software package. This tool will configure the CMP EJB to run in most commercially available EJB servers that support container managed persistence EJBs and coordinate the database access with the CocoBase runtime libraries.

Clicking Next > in the Generic EJB Entity Bean CMP—All Parts drop down wizards of CocoAdmin will present a user with a list of attributes which comprise the maps and any references (e.g. foreign keys) to other maps that may be navigated by CocoBase. For example, in a e-commerce shopping cart example, a Customer map, which is generated against a selected relational database connection, might be selected, and the PkgName: field might contain the name testpkg so the resulting Java code will be generated to a package and directory named testpkg. In a subsequent pop-up dialog, the user can add a foreign key reference by clicking on Insert Attribute and filling in the foreign key attribute in the inserted row. Such references can be added automatically for any operation which contains a join across tables of different maps. This topic is covered in more detail in the CocoBase Programmer's Guide.

Customizing Attributes to be Generated in Java Code by CocoAdmin

Attributes that are to be generated into the Java code by CocoAdmin can be customized through the code generation screen of CocoAdmin, which is displayed when a map is selected from the wizard connections tree and the Next > button is pressed.

For each fixed Map Attribute Label in the leftmost column of the table corresponding to a selected map, the corresponding Java Attribute Name and Field Type are editable. When the Field Type field item is placed in edit mode, the CocoAdmin user can select one of the available data types from the drop-down list. These relational data types will be mapped to a corresponding Java type when the code is generated. A relational to Java conversion table can be found in CocoBase Programmer's Guide, but any functional conversion table may be used. If foreign key references are present, each entry of the reference is also editable.

In general, the attributes for Java code to be generated should not be customized here unless a foreign key relationship needs to be modeled that isn't described by the database foreign keys or that isn't described correctly. Database foreign keys are automatically included in the map if the ForeignKeys checkbox in the wizard connections window is checked. Editing a map to include foreign key relationships is covered in the CocoBase Programmer's Guide.

A CocoAdmin user can specify a key for the map for the purposes of code generation by checking the box under the Key column. The attribute corresponding to the checked key box will be used as a key for the map and more than one box can be checked. For non-EJB applications, it is not necessary to specify a key, but keys can be useful in applications such as object caching and should contain unique values. Because EJBs require a unique key (for example, see the EJB spec. 1.0 from Sun Microsystems), CocoAdmin will not let a user generate code for an EJB without first specifying a key.

A Customizing an Output Filename for Generated Java Code

When a map is generated into Java code by CocoAdmin, the output class name is specified manually by entering it in the File name field. If no filename is entered, the user will receive an error message when the user attempts to generate the Java code. Generally the user should adhere to Java naming conventions when naming package components. Package components should be saved in the PackageName directory and Java class files should have the form Classname or ClassName.java when entered into this field.

A CocoAdmin user can specify an existing output directory for the generated class (generated code) from the Look in: drop-down dialog. If from a previous window, the user specified a package prefix for the source code, then a subdirectory matching the prefix name can automatically be created in the specified output directory, if one does not already exist. Further information on naming and naming customization is presented in sections of the CocoBase Programmer's Guide.

Generating Java/EJB Code Using CocoBase

Once a filename has been entered as described above, a CocoAdmin user can generate the Java code by clicking the Generate button. When the Java Class or EJB has been generated, a dialog box indicating a successful generation will be displayed. If a user attempts to generate an EJB without specifying a key attribute, an code Generation Exception Key must be specified! error message is received. Likewise an error message may be received if a file already exists with the filename that was specified by the user for code generation.

Prior to code generation maps of CocoBase may be customized to add relationships between fields, objects, and beans by creating links. Relationships between objects such as 1 to 1, 1 to many, and many to many may be set. Among other relationships, parent child class relationships are supported. Code may be generated which reflects such relationships. Also, during the code generation step relationships may be added using wizards and interfaces of CocoAdmin.

Link information can be dynamically entered and edited with a CocoAdmin dialog, or it can be loaded from the

resources\cocoNavLink.properties file of CocoBase if code regeneration is going to occur often with the same class, and if keying in virtual foreign keys isn't practical. The database foreign keys are auto-detected at a table level, and can be edited using a CocoAdmin dialog if the Map name to be used is different from the default one generated using a table name.

Customizing Generated Code Through CocoBase Templates

The later versions of CocoBase Enterprise all have the ability to customize code generation through the use of multiple code generation templates. Templates are available from the tool through the CocoAdmin.properties configuration file. Each of the GENTEMPLATES name values found in a configuration file are comma delimited and specify the name of the code generation template to be used. The comma delimited file can be viewed using an ASCII text editor, for example. The "\" character which appears at the end of each line indicates a line continuation and is only relevant to the CocoAdmin tool. Name values present in this file can also describe a set of templates that will be processed simultaneously. The code generation list that appears in the tool by default includes, among other things, all of the major Java Application Server types, production of generic CMP, generate Java classes, generate JSP, and the like. See the current www.thoughtinc.com website for a more complete listing. The default Java object of the listing is a standard Java instance that implements the CocoBase interfaces. If a GemstoneJ EntityBean using Proxy objects is selected, for example, CocoAdmin will generate all of the GemstoneJ files necessary for a Bean Managed Persistent EJB Object. Such an "All Parts" template reference will actually cause several files to be generated. Such a selection would have a Cocoadmin.properties entry that specifies which template files are to be processed, and in what order to process them. Such a file might be found in the thought\cocodemos3tier31\demos\resources directory of a particular version of the installed CocoBase software package. In one embodiment of the invention, when a particular template is picked any file prefix/suffix values to be appended to the filename would also be prefixed or appended to the map name automatically.

Mapping from One Data Source Format to Another Data Source Format

In another embodiment the invention provides a system for mapping from a first database format to a second database format, or from one database to another database of the same type, as a way for transferring data or synchronizing data sources. The system includes: data in the first database format stored in the system; rules for translating from the first format to the second format stored as a separate structure from the data; and means for applying the rules to the data to obtain the second format. This system may include a means for transferring the data from a first data source to a second data source. In one embodiment this system includes a simple computer program in a computer language such as Java, which reads the data from the first database using a repository map that may optionally be cached in the computer memory of the system and then stores the accessed data to a second database using the same or a different repository map.

An example of the structure for such a system and a short description/outline for a computer program adapted for such a system that is capable of transferring data between a first data source and a second data source might be described as follows:

First, let us assume two databases: a first database (B1) and a second database (B2) wherein B1 and B2 have the different corresponding data map schemas (S1) and (S2), respectively, and wherein S1 and S2 have the two corresponding schema repositories (R1 and R2) that include maps defining the structure (schema) of the database, Java object information (Java object model and any relationships between Java objects of the Java object model, or defining both the database and Java object information.

M1 is a map (or maps) defining the database schema S1 of R1 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B1, and

M2 is a map (or maps) defining the database schema S2 of R2 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B2.

The sample program outline,

(i) open a first system connection to access R1 and read M1

(ii) open a second system connection to access R2 and read M2

(ii) have a Java application create a Java object (Obj-1) in memory that can access the data of B1

(iv) allow the Java application to connect to B2 and access B2, or have the Java application create a Java object (Obj-2) in memory that can access the Obj-1 and also access B2,

(v) open B1 and B2

(vi) have Obj-1 retrieve data from B1 and place data in Obj-1

(vii) optionally have Obj-2 obtain data from Obj-1

(viii) have the Java application access B2 (optionally through Obj-2) and store the data of Obj-1 or Obj-2 in B2

(ix) clear Obj-1 and possibly Obj-2, or create a new instance of Obj-1 and possibly of Obj-2

(x) repeat steps (vi)–(viii) until all of the data is transferred from B1 to B2

(xi) close B1 and B2 and any other open files, and

(xii) end program.

In one preferred embodiment, step (vii) as described above involves having data retrieved by Obj-1 and then converted to an XML format data file which is then forwarded to a different location (and optionally simultaneously to a different user) on a distributed network, whereupon an existing or new instance of Obj-2 is populated with data from the forwarded XML format data file and Obj-2 then stores the data in B2. The XML format data file may be simply of a Java object, the data than can be stored in a Java object, or both the Java object and its data.

The above procedures implement a dynamic mapping layer in an exchange format such as XML that can directly exchange data from two data sources in a more dynamically controllable fashion to each other for the purposes of data import, export and exchange without the requirement of an intermediate third object. In such a model the developer would not be constrained by the structure of the database to which or from where the data is being transferred.

Also, the above example is an example of object to "DataSource" mapping. Data may be mapped to (or from) object and relational databases and may also be stored in XML. Also, XML stored maps can be utilized by CocoAdmin as a resource from which to generate object code for user applications. In one implementation, two system users (same or different system) using a single (or several major recognized) XML standard(s), can export or import (as XML files) either or both of the XML map definitions

(relationships of data) and data (dataset) of the database. This would permit a user to distribute parts or all of a dataset (and/or maps) to a distributed network, or to multiple network users in useable XML format files. Such a phenomenon may be referred to generically as O/X mapping (object to XML) or R/X (relational to XML) in addition to O/R (object to relational mapping).

In one embodiment, O/X mapping step would permit importation or exportation of data (that was converted to XML) from XML to and from any business object. For example, data from any business implementation object could be exported as XML along with its relationships as defined by the mapping tool. This would implement a more flexible and dynamic mapping facility for taking XML datasets and using them to populate object instances other than the original ones that may have created the data. One good example of such an implementation would be to take a first XML document that is presented as a user screen interface to a dataset and utilize the information (dataset) populating the first XML document to seamless populate a second XML document (datasource to datasource via a XML (or another exchange format similar to XML) translation step.

There are many ways to utilize the above above implementations as part of either uni-directional or bidirectional mapping. Intranets and e-commerce applications can both take advantage of these important features. There are distinctive advantages when using a dynamic mapping layer where a java object provides translation by mapping such objects of a first data source (relational or object database) and also mapping such object to an XML or other second format data source. This allows more developer control as to how datasets are exchanged, filtered and/or validated between a first data source and a second data source.

One business use would be for a first party to use XML as a data export document to send information to a second party (e.g., items offered for sale) and the second party would import that information in order to generate a second XML data document (e.g., purchase order) that the first party can import. This would provide a much needed exchange format for the newly developing business to business market that might be able to rely on standardized XML data formats and avoid problems caused by object programming models that may vary widely from company to company.

Such a dynamic mapping facility would provide a critical missing facility to easily translate data models and object model into a format that can be readily used by business to business companies, or by the same company for department to department interactions.

Documents describing the use of CocoBase products that do mapping of data sources other than relational databases have been published, and such products may be used in conjunction with the above embodiments. Examples of mapping include, mapping from object to object databases and from COBAL (mainframe computers of businesses often use this language data source) to object. See the IBM "white paper" on the topic of COBAL to object mapping (may be downloaded from URL http://www.thoughtinc.com/websphere.html).

In one embodiment of the system of the present invention a translation layer translates between an object application (or a potential object application, i.e. an object model) to at least one relational database which includes data entries organized as tables and records. Examples are database architectures supported by companies such as Oracle, Sybase, Informix, etc. Such an organization is well-suited for manipulation by relational query languages such as SQL. However, the traditional relational database organization is not ideally suited for manipulation by an object-based system. Some tools, such as Java Database Connectivity (JDBC) exist to achieve some degree of indirect relational to object mapping by accommodating some differences between an object model or application and the structure of the relational database. However, these tools also have drawback in that they are often not scalable, require extensive manual recoding for different object applications and are complex to use. Even tools (often mistakenly referred to as object to relational, i.e., O/R tools) that generate JDBC code (instead of generating a pure SQL string that a JDBC driver can utilize to subsequently create a pure SQL statement or query bundled in a JDBC format) are limited since they are often application specific, database specific, or specific to both.

In one preferred embodiment, a translation or abstract layer communicates with at least one JDBC (relational database driver) and at least one primitive Extended Java Bean (EJB) construct. The function of such a translation layer (generally called the O/R layer in CocoBase documentation, for example) is to translate object-based queries for the data into queries that JDBC can, translate into queries for a relational database. In a preferred embodiment, the translation layer can generate an SQL string (or strings) based upon the object-based queries, which can be passed to at least one JDBC, which JDBC can then generate an SQL statement from the SQL string. Similarly, the abstract layer accepts results from the queries and provides them to one or more of the EJB constructs in a suitable object format. The existence of JDBC is not necessary for all implementations of the invention. Also, different types of databases, data models, computation architectures, and the like can be used within the basic principle of the present invention. This mapping can be one-to-one, many-to-one, many-to-many, or any variation.

Another preferred embodiment of the present invention allows for mapping tables to be plain text, or to be text-based XML repository files. This not only allows the maps to be human readable and editable with standard editions, email programs, web browsers, etc., but allows for easy transportability and delivery of the maps.

Another feature of the maps is that they can be dynamically loaded into a computer system to provide for new dynamic mapping in a system that can that can continue to run while the new maps are loaded and executed. As described supra, the use of the translation maps provides advantages in organization, standardization of interfaces, efficiency of design, compatibility, scalability, portability and other advantages. This translation system of the present invention provides an entire set of tools to simplify and improve an operators ability to manipulate the maps without requiring a high level of programming knowledge or database query language familiarity. in part, the present invention is based upon a translation operation derived from the operation of U.S. Pat. No. 5,857,197, and depicted by the flow chart at FIG. **1**. At step **1**(*a*) the application software is operating, and reaches a point where data must be accessed to continue the operation. For purposes of example, the application software is object oriented, and is being run by a customer at a customer location. However, the application software need not always be limited to object orientation. Rather, other types of software models can be facilitated by the present invention to be described herein.

Once the application software reaches a point that data is needed, for example, a production database or data store, which is not at the site of the application or controlled by the

application user, then a request **2**(*a*) is made for that data. As described in great detail in U.S. Pat. No. 5,857,197, when an object application makes the request for data, certain object information is sent to a location whereby the necessary data can be accessed.

The object information provided the object application is received in an abstract layer, also known as a translation layer, at step **4**. At this point, an analysis takes place to extract certain information regarding the object sent by the object application. This information is used to select and manipulate a particular metadata maps provided by a data manager, which handles the data store containing any information required by the object application.

Operating entirely independent of the application software is the data manager. The data manager carries out an analysis of the arrangement of data in a data store controlled by the data manager at step **1**(*b*). Depending upon the size of the data manager and the data store, any number of analysis can be carried out to help facilitate access by parties interested in the data held by the data store.

The result of this analysis is the development of a metadata map or maps based upon generated object characteristics of the arrangement of the data store. Depending upon the size and arrangement of the data store, any number of metadata maps can be generated. The number of generated object characteristics, which would be the basis of the metadata maps depends upon the data manager operator, and his/her response to the requirements and characteristics of those seeking access to the data store.

At step **3** a metadata map or metadata maps, based upon the object characteristics of the arrangement of the data store, are generated. The number and characteristics of these maps depend upon the characteristics of the arrangement of the data store, and the responsiveness of the data manager operator to the various types of demands for data by object applications. Each metadata map provides the necessary means to obtain data or particular types of data from the data store. It should be noted that in many cases there are many object characteristics can be derived from a relational data store. The data related to particular derived object characteristics that can be accessible using a metadata map generated for a selected group of object characteristics.

The operation of the object application and the management of the relational data store come together at step **5**. At this point, an evaluation is made between the object information provided from the object application and object characteristics associated with the metadata map available in the data manager. The evaluation results in a selection of at least one metadata map appropriate for the object characteristics sent from the application. The selected metadata map is used to navigate through the data store to obtain data that is desired for the object application at step **6**. It should be noted that more than one metadata map can be used to obtain data from the data store.

Once the data is obtained, it is sent back to the requesting object application at step **7**. It should be noted that as described in U.S. Pat. No. 5,875,197, step **6** of obtaining data and step **7** of sending data are constituted by a much greater number of sub-steps. These include: generating demands from the metadata map; executing the commands used by the metadata map; obtaining data from the data store in response to the command; processing the data access from the data store using the metadata map; packing and manipulating the data for transmittal; unpacking the data; instantiating objects using the accessed data and returning the instantiated objects to the object application for further processing. It should be noted that the results of the overall data accessing process may contain many rows of data which can be used to instantiate a plurality of objects.

An important purpose of the aforementioned system is to achieve a high degree of flexibility with dynamic mapping to objects as they occur in object applications. Accordingly, it may become necessary to handle a plurality of objects at any one time, and obtain the necessary data to instantiate or otherwise satisfy requirements of each of the objects addressed in the object application. However, large amounts of data associated with each object may become cumbersome. Further, a single map for plural objects may also become awkward to handle.

Evaluation of object characteristics sent by object applications can often be time-consuming while not resulting in the selection of the best metadata map, and thus the most appropriate data may not be accessed. Further, depending upon the amount of data available in a data store and the number of metdata maps available, the selection process may become very slow and cumbersome.

However, there is a definite need for translating between a wide variety of different data formats, other than just the relational to object translation of the basic system of U.S. Pat. No. 5,857,197. There is also a need for transmitting quickly and easily between various types of databases. Further, there is also a need for editing or otherwise altering data exchanged between databases for the purpose of facilitating object applications. The basic system of U.S. Pat. No. 5,857,197 needs further support to facilitate expeditious operation, both for its basic system and for modifications to its system that will accommodate better security and for selectivity of data.

The enhanced operation and efficiency of the basic system by U.S. Pat. No. 5,857,197 is enhanced by the inventive modifications depicted in FIG. **7**. The goal and purpose of the modified operation of FIG. **7** is to more quickly handle greater amounts of data by streamlining or otherwise enhancing the operation depicted in FIG. **1**. The operation of FIG. **7** also facilitates greater flexibility in that the system more easily handles multiple maps for a single object as well as use of plural objects for a single map. The reconversion process of FIG. **7** also allows conversion between database formats of a wide variety. As a result, editing of data for a particular object application (or other type of application) is more easily facilitated. Further, changes in data for a particular object need be done only once if such changes are handled in accordance with the invention depicted in FIG. **7**. A key attribute of the process of FIG. **7** is the enhanced processing speed facilitated thereby.

In the system of FIG. **7**, steps **1**(*a*) and **2**(*a*) are carried out in the same manner as in the process of FIG. **1**. It should be noted that the invention of FIG. **7** does not require that only objects be used to obtain data from relational databases. Rather, virtually any type of database can be accessed by virtually any type of application. However, for purposes of presenting a coherent example, translation relational database to objects required by an application will continue to be used. Also, as with the FIG. **1** embodiment, an abstract or translation layer is used to receive the request for data related to the objects provided by the object application. It should be noted that the abstract layer, which carries out much of translation activity can be constituted in a number of different ways. For example, the abstract layer can be the computers and communication links of a third entity (in addition to the application entity and data manager entity). One example would be the business entity Thought, Inc., the Assignee in the instant application. Thought, Inc. would provide all of the necessary software for the various trans-

lations and other operations that would provide the translation of relational data into object oriented form required by the object application in a manner that appears to be transparent.

However, the abstract layer need not be confined to a single business entity for a single group of computers. Rather, the functions of the abstract layer can be distributed over a number of different entities and associated computers, linked via the Internet, a number of different intranet arrangements, temporarily dedicated lines for fax or FTP transfers, satellite links, microwave links or any other communication link which will be used for exchanging data.

In another alternative, the abstract layer could be associated with the data manager, and be operated as a part thereof. However, depending upon the scope of data handled by the data manager, operations of the abstract layer could be limited thereby. In yet another alternative, the abstract layer would be part of the operating system of the entity operating the object-oriented application of software. However, the same limitations would apply thereto, as indicated for the situation with the abstract layer was part of the data manager. Clearly the most flexible arrangement is for a separate entity to handle the abstract layer, providing access to a wide variety of different customers operating these types of application software, and providing access to a wide variety of data stores and data store types. It should be noted that growth of such arrangements with many application user customers, accessing a wide variety of databases, leads to certain problems of scale. Many of these are addressed by the further system operation depicted in FIG. **7**.

Rather than going through the process of extracting object data for conventional evaluation at step **5**, this cumbersome and often slow operation can be bypassed by a cache check operation (step **51**). While the cache check at step **51** can be carried out in the abstract layer as indicated in FIG. **7**, the present invention is not limited thereby. Rather, the object data can be also be contained in a cache to be checked at step **3**(*a*) at the application site. In this manner, the necessary object data can be transmitted directly to the abstract layer and used for the evaluation with the metadata (step **6**) without going through the cumbersome extraction process.

It should be noted that the cache check operation **51** can be carried out by a plug-in software addition. In the alternative, the abstract layer may act as another database at a remote location. Apparently, the best method is to keep a cache memory located within the processing computers of the abstract layer to more quickly facilitate the cache check **51**. This allows the manager of the abstract layer to control the object data that is cached for easy re-use.

In a preferred embodiment, the present invention to provides a fully synchronized caching system that utilizes the transaction coordination facilities of a server such as a J2EE application server (for example, Weblogic 6.1), such system comprising a local or distributed computer system having a first data source referred to as the primary data source, a second data source referred to as the cache data source and the cache data source is associated with an object to relational mapping layer to provide a data source cache for object applications, and a server having a transaction coordinator with the ability to register multiple data sources, wherein:

(a) both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and

(b) the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and

(c) registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be sycronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

A number of primary object databases exist in the art. Non-limiting examples are: (a) Gemstone by Gemstone, Inc., Beaverton, Oreg., (b) Itasca by IBEX Corporation, Minneapolis, Minn., (c) Jasmine by Computer Associates International, Inc., Islandia, N.Y., (d) Matisse by ADB, Inc., Redwood Shores, Calif., (e) NeoAccess by NeoLogical Systems, Berkely, Calif., (f) O_2 by Ardent Software, Versailles, France (g) Objectivity/DB, Objectivity, Inc., Mountain View, Calif., (h) ObjectStore and Javelin, both by Object Design, Inc., Burlington, Mass., (i) ONTOS DB by ONTOS, Inc., Lowell, Mass., G) POET by POET Software Corp., San Mateo, Calif., (k) UniSQL by Cincom Systems, Cincinnatti, Ohio, and (l) Versant by Versant Object Technology Corp, Menlo Park, Calif.

A number of primary relational databases exist in the art and have been described earlier in text that was presented above.

A number of SQL searchable memory resident relational databases that utilize JDBC drivers exist in the art, which may be adapted for use as a secondary database buffer in the context of the present invention. Non-limiting examples are: (a) HypersonicSQL, an open source relational database (publicly available for free at http://sourceforge.net including source code) that is simultaneously disk and memory resident, (b) TimesTen by TimesTen Performance Software, Mountain View, Calif., and (c) Polyhedra by Polyhedra, plc, Bellview, Wash.

A particularly preferred embodiment of the invention provides a cache data source can be set up as a memory-resident entire database, a disk resident database, or both a memory resident database and disk resident which are synchronized. In one object, of the invention the memory resident database may be a portion of the disk resident database and the size of the memory for the memory resident database can be set by the user.

Below is a general example explaining how a plug-in can be used as a database synchronization mechanism across multiple databases such as across a first database and a second database.

Example and Explanation for Using the CocoBase Plug-in Architecture to Synchronize Heterogeneous DataBases

Suppose the existence of 2 databases, **B1** and **B2**, with different schemas, **S1** and **S2**.

Assume both **B1** and **B2** have their own CocoBase repositories **R1** and **R2**, where **M1** is a map defined in **R1** and **M2** is a map defined in **R2** and both **M1** and **M2** are optionally identified by the same map name.

Suppose that the system uses CocoBase (commercially available object to relational mapping tool having a mapping repository capability) and a connection of CocoBase with **B1** is established as follows:

CocoDriverInterface connB1 =
CocoDriver.getCocoDriver(cbdriver,jdbcdriver,jdbcUrl +
";cocoplugin=thought.CocoBase.SynchPlugin",user, passwd);
"thought.CocoBase.SynchPlugin" is the name of the plug-in class that is configured to replicate operations of database B1 to database B2. For example, for the "insert" operation, the plugin implementation would be similar to the following

-continued

```
(suppose replicationDB is set to connection to database B2):
    public int insert(int code, CocoUser userObj, Object
keyObject, String objectName, Integer objCount) {
        switch (code) {
        case CocoServerPlugin.CB_PRE:
            break;
        case CocoServerPlugin.CB_POST:
            Object o = null;
                if (objCount.intValue( ) >= 1) { // if object
was insert in B1
                    this.replicationDB.insert(keyObject,
objectName); // we insert it in B2
                }
            }
            break;
        default:
            throw new RuntimeException("Invalid plugin!");
        }
        return CocoServerPlugin.CB_CONTINUE;
    }
```

Once the above connections of CocoBase are established, any accesses to the database **B1**, such as:

connB**1**.insert(. . .)

will have the plug-in access **B2** and replicate the operation that was conducted on database **B1**.

The same mechanism described above can be used to replicate the result of other database operations (e.g. update, select, delete, etc.) that are performed against **B1**. Further, the application that uses CocoBase maps to access **B1** and **B2** can have CocoBase set **B1** as the primary database and **B2** as the cache database. In such a situation only the cache database **B2** needs to be accessed to retrieve data when data that the application needs is present in **B2**, but both data sources need to be accessed in order to insert or store data. This can significantly speed up data retrieve by avoiding disk accesses for data already present in a memory cache.

In the case where **B1** and **B2** are both registered as data sources in an XML descriptor file with a J2EE application server (for example, Weblogic 6.1) in combination with the CocoBase plug-in cache facility, a fully synchronized caching system can be provided that utilizes the transaction coordination facilities of the server. Very important and significant database caching abilities along with transparent data persistence is provided while providing transaction coordination that is controlled by the J2EE application server.

To more specifically describing the cache check **51**, in virtually any situation involving the cache check **51** of CocoBase, separate transactions will be carried out for cache check operation. In such a transaction a check of the cache will be made. If found, the relevant object data will then be moved to the next step of the operation for evaluation with the metadata (step **7**). If, on the other hand, the relevant object data is not found, the evaluation process takes place. This is controlled by the abstract layer manager for purposes of the present example, and includes an evaluation of the object data to determine what should be placed permanently in the cache memory that is used by the abstract layer for containing object layer data. Such an evaluation can be carried out for adding object data to a cache, whether or not the cache is controlled by the manager of the abstract layer. For example, it is entirely possible for data manager to carry out such a caching operation more easily supplying those metadata maps that are requested most often.

In the parallel activity occurring at the data manager, steps **1**(*b*), **2**(*b*) and **3**(*b*) are carried out in the same manner as the operation of the system in FIG. **1**.

The difference is that at step **4**(*b*) selected metadata maps are converted to the XML format. This is a format already known for its facility in the handling text, thereby making modifications to the text of the map relatively easy. Selected metadata map or maps are loaded at step **6** into the abstract layer. The evaluation process at step **7** takes place to determine how to construct the actual commands to access the necessary data from the data store in order to satisfy the request from the object application (step **8**) in XML format.

The use of XML format facilitates modification of the map with simple text editors. Once the map is altered (to determine how data is to be stored and arranged), the application user can then use the map to effect the desired data handling for any associated objects. As a result, the data is easily manipulated without extensive recoating or programming expertise for the data store architecture, as is required with conventional systems.

Another alternative that speeds the overall operation of the translation process is to convert selected cached objects into XML format at step **52**. In effect, the system is carrying out objects to XM mapping to facilitate faster operation to the overall system, as well as easier modification of the object and map.

Based upon the presumption of a future single (or several major) XML standards, it would be practical to export not only XML map definition but also the production data accessed by the map definitions. Once formed, the data can be easily distributed to an entire network. Thus, step **8** of the FIG. **7** embodiments could be further modified so that the data is sent in XML format rather than just the map. XML is a highly desirable data exchange format since the data can be e-mailed to XML format more easily. The speed of translation and evaluation discussed for the FIG. **7** invention enhances the dynamic operation of the overall system in that maps can be loaded very quickly, responsive to requests based upon object data. Further, the map can also be altered dynamically (during the run time of the application) to suit the requester managing the application software. Further, the object to XML mapping would permit importation of data from XML format to any business object. Likewise, data from any business implementation object could be converted and exported in XML format along with its relationship, as defined by mapping tool such as metadata map.

Greater flexibility and faster dynamic mapping can also be achieved by taking XML data sets and using them to substantiate selected objects. An example of this would be to take a first XML document and utilize the information (data) within the first XML document to seamlessly substantiate objects in a second XML document. While this can be done by altering step **4**(*b*) in FIG. **7** to convert data store to XNL in response to a provision of selected objects.

Accordingly, there will be a substantial advantage in generating XML data documents for exchange between parties to carry out commercial transactions. Conversion to XML will also facilitate the use of data sources or data architecture other than relational databases. Examples of such mapping includes object-to-object mapping and JAVA to object mapping. Once algorithms translating between XML and object formats are standardized, translations between any type of system can take place seamlessly and very quickly. Mapping can also include the use of JAVA Server Page (JSP) format.

Once editing has took place either for the metadata maps or even the production data from the datastore, changes can be recorded in the operation of "snapshots" so that a record of the map or other data at various times can be studied. As a result, data management paradigms can be developed.

Normally such "snapshots" are made at the location of the party requesting the data (such as the object application user) once map modification is completed. The modified filed can then be placed in a source code management system.

The snapshots or map modifications, or even general data of modification is facilitated through the use of caching. Since a variety of different types of data (including metadata maps, objects, production data, and the like) are used as options in the inventive system, a number of caching methods must be adopted in order to accommodate the various types of data. Examples of caches include: a write-through cache without transactional integration; a two phase commit integrated cache; and, a temporary working storage cache, which acts as a second database. All three of these techniques are discussed in more detail below. The present invention is also used to distribute caching or database using a plug-in Java database. Any number of different caching arrangements can be applied to the system of the present invention, but are limited only by what is possible in the art of data storage. The various caching arrangements can also be used to facilitate global changes in maps and data.

If caching arrangements are to be used with CocoBase, the accessing and updating of information must be defined in some clear way to control the caching and avoid unclear or unacceptable results. Discussed below are three methods for our caching controlling, followed by two examples of ways we cache in memory as either a "database" in memory with a third party plug-in or a "caching in memory of data" where both are connected to the O/R run-time layer. When a user connects to the O/R layer and subsequent database they register any optional plug-ins for caching or extending behavior and this then controls the access to data per one of the three mechanisms. Such options can be very powerful when combined with transaction capabilities of some Java application servers.

Write-Through Cache Without Transactional Integration

The first type of cache design is a "write-through cache without transactional integration" (no two-phase commit). This model assumes that applications not going through CocoBase may also be updating data in the first database (data source). In this method if a store data attempt fails at the first database, then the corresponding data of the second database (cached data) is deleted and may be (or may not be) refreshed from the first database depending upon developer policy. If the store data succeeds at the first database, then the corresponding data is updated in the second database (cached data). Notification of storage attempt results can be sent to the user to indicate a successful or failed attempt to store data. Therefore, multiple users can be utilizing the same cache in this instance and multiple applicationw (e.g., applications that do not go through CocoBase) may be accessing the first database and data integrity can be maintained.

Two-Phase Commit Cache

The second type of cache design is a two-phase commit cache design. This type of caching only allows updates of data by applications that work through CocoBase, through an application(s) that is either a compatibly integrated application(s) working concurrently with CocoBase, or uses the same policies of updating as CocoBase. In this cache model, data is only updated if CocoBase, or the integrated application can update the data in both the first database (source) and the second database (cached data). A commit to change data in both the first and second database (or information cache) is obtained as a first step and store data (update) completion is done in a second step. Again multiple users can use the same database source (or multiple application if they are integrated properly with CocoBase) and data integrity can be maintained.

Temporary Working Cache

The third type of cache design is a temporary working cache design. This type of caching only allows updates of data by applications that work through this temporary working storage cache (i.e., memory buffer or a second database which may be completely or partially memory resident database). This temporary working storage cache is designed to shield a first database from access and act as the datasource to users. The updating and accessing of the first database can be defined by rules of the developer as they desire. This cache design assumes that database accesses by database information users are directed to the cached information through CocoBase or through another application. The information in this working storage cache is used to intermittently update or to not update the first database (data source) as defined by the developer. In such a model, transactional coordinated database updates and persistence should be synchronized by registering both databases with the application having transaction coordination, such as the J2EE server Weblogic 6.1, for example. In such a case, this cache design may be described as a synchronized transactional database caching model.

The CocoBase software suite provides distributed caching of a database by using a plug-in third party free java database (for example HYPERSONIC SQL), however any third party java accessible database could be used. For example, a CocoBase application user can define a map of a first relational or object database to be accessed, and then import the data into a memory cache through a plug-in java accessible database (second database). This can allows CocoBase or compatible integrated applications to control access to data as defined by a developer, and database users may re-directed to access the second database in memory instead of accessing the first database in order to speed up performance. This also permits a database to be distributed to multiple users or to multiple applications on a network, where CocoBase or pre-defined developer rules control the updating and persisting of data by storing back to the original database any changes made to the memory cached java accessible database according to one of the three caching methods described above.

A distributed cache may use optional mapping server plug-ins and control access to multiple server databases. In plain English, the data source object database such as Oracle is opened as a first server database, and a second server database (for example HYPERSONIC SEQUEL) is also opened as a third party memory or virtual memory resident object database. The second server database can be loaded into a designated mapping server memory area. Once the first database is accessed, the data is sent to the second server database for storage and also sent to the user. Future accesses to that same data are directed by CocoBase to the second database. Also, updates of data are directed to both the first and second database by CocoBase and/or by a transactional coordinator of a J2EE server such as Weblogic 6.1 where both databases are registered with the J2EE server.

Alternatively, some of the latest alternatives for caching use the caching capability of a server to which CocoBase is providing maps and connectivity with a database, or the caching capability of another accessible caching facility. Any stand alone caching facility that a developer connects with the O/R layer as describe above in the caching methods descriptions can be implemented as a cache for database accesses).

A caching facility may be utilized by CocoBase to create a single or multiple objects in accessible caching facility memory, which object (or objects) may be stored in a relational database (or really any other piece of data, data object or data stream) object database, etc. for possible re-use or multiple use by application(s). CocoBase, integrated applications, or pre-defined developer rules will then control access to data and can permit the developer to require a user to directly access the data in cached memory rather than accessing the original data source. This can really speed up performance, e.g., up to 40 times faster with the combination of Oracle as the first database and Hypersonic as the second (cache) database than by using ordinary Oracle database accessing.

For optimizing performance of applications that utilize the CocoBase runtime libraries and other repositories, it is important to cache not only data, but also to cache the map (or maps) to be used in the CocoBase mapping runtime. The underlying mapping connection automatically does this in CocoBase. So for example if a user is accessing an instance of 'Customer', the underlying thought. CocoBase. CocoPowder software module will cache the map information for the customer map within the connection object.

In terms of CocoBase, map caching is when the software suite reads and caches the map created with a mapping administration tool such as CocoAdmin into RAM (random access memory) of the computer system. Once this map is cached, the CocoBase runtime repository libraries can generate and execute SQL much faster, but the first step of reading a map into RAM can be relatively expensive resource-wise. The cached maps are kept internally by CocoBase in a pre-compiled form that is exceedingly efficient for the runtime to access. Maps by default are cached within a connection once they are loaded and used for the first time. It is also possible to cache maps for all connections within a Java Virtual Machine (JVM) by using the CocoBase URL extensions for controlling cross connection caching.

There are two mechanisms that can be used currently with the CocoBase enterprise runtime libraries to control map caching.

The first is to use a cache configuration file that specifies which maps to pre-fetch and to place in shared cache. This type of map caching is controlled through a syntax such as:

JDBCURL:cocorep=configresource:/resourcefile.propertieS or for example

JDBCURL:cocorep=configfile:/opt/file.properties

The cocorep syntax can begin with a colon ':' or semi-colon ';' syntax, so it can be compatible with different computer systems. The resource syntax describes what config resource should be loaded from the CLASSPATH, either directly from the file system or from some archived file such as a JAR or Zip file. The syntax of the Config file is that a list of MAPNAMES needs to be specified in a comma delimited form for the tag MAPNAME. A sample resourcefile.properties file might appear as follows:

#Map names to preload

MAPNAMES=Customer,com.mypkg.Address, com.altpkg.Order

Another type of CocoBase software suite map caching that can be done is based on an XML repository. A set of maps can be exported as an XML repository and either placed into an archived deployment file, such as a JAR or zip file, or the XML repository can be an un-archived file that is loaded directly from the file system. To specify an XML repository in the CocoBase URL, the syntax would look something like:

JDBCURL;cocorep=resource:/CocoRepository.xml

or

JDBCURL;cocorep=file:/mydirectory/CocoRepository.xml

Both the colon and semi-colon are supported as the separator character for CocoBase URL extensions. As a result '; cocorep . . . ' and ':cocorep . . . ' would be processed the same by the CocoBase runtime libraries. Some EJB descriptor syntax requirements place special meaning on the colon or semi-colon, so CocoBase makes it possible to use either of these two syntax choices.

Fortunately, map caching with CocoBase only needs to occur once a session or once a JVM depending on the option used, and then it is cached in RAM. This initial read and caching process should be taken into account if any performance tests are being created to test CocoBase. Using the per JVM map caching can be a great way to pre-load and pre-fetch maps so that performance numbers are accurate based on normal execution patterns.

The mapping server of CocoBase can be extended through custom developed classes called plug-ins. A CocoBase plug-in implements the Java interface thought.CocoBase.CocoServerPlugin. This is a standard Java interface, and any Java class, which implements it and provides a public default constructor can be loaded automatically by the server at startup. Examples of such plug-ins are the cache and connection pool that ship as source code in the demos directory of the CocoBase software tools suite.

A CocoBase mapping server plug-in will implement the following interface signature:

```
public interface CocoServerPlugin {
public int call(int code, CocoUser userObj, int objCount, Object keyObject, String objectName, Vector returnObjs);
public int close(int code, CocoUser userObj);
public int commit(int code, CocoUser userObj);
public int connect(int code, CocoUser userObj);
public int delete(int code, CocoUser userObj, Object keyObject, String objectName,Integer objCount);
public int getColumns(int code, CocoUser userObj, String catalogName, Vector colInfo);
public int getValue(int code, CocoUser userObj, Object key, Object value);
public int insert(int code, CocoUser userObj, Object keyObject, String objectName,Integer objCount);
public int rollback(int code, CocoUser userObj);
public int select(int code, CocoUser userObj, int selectType, int selectCount, Object keyObject, String objectName, Vector returnObjs);
public void setCommObject(Object comObject);
public int setValue(int code, CocoUser userObj, Object key, Object value);
public int update(int code, CocoUser userObj, Object keyObject, Object newObject, String objectName,Integer objCount);
}
```

In such an implementation, each method will be called before and after the associated operation (such as select) and the code will specify whether the action is a pre or post process operation also defined as public static codes in the plug-in interface API class.

```
public static int CB_PRE=0;
public static int CB_POST=1;
```

The Post condition will always be called unless the plug-in aborts or finishes the operation in the pre-condition of the plug-in. Otherwise the plug-in will finish its pre

processing, continue into the actual mapping connection, and then call the plug-in again with the post processing flag. After the completion of the post-processing, the resulting Objects or return codes will be passed back to the client process.

If the plug-in wishes to intercept and 'finish' an operation in the pre-process state without actually having the Mapping layer become involved, then the CB_FINISH return code can be returned as the return code of the method. If there is an error in the pre-processing and it is desirable to abort the request once again without involving the actual mapping layer, then we can return a CB_ABORT code. Otherwise the CB_CONTINUE should be returned (an all is well code).

```
public static int CB_CONTINUE=0;
public static int CB_ABORT=1;
public static int CB_FINISH=2;
```

Since all select operations come through the single 'select' callback, a software application may need to know which method the client called. CocoBase uses a code to distinguish what the client actually requested the call with the following values. This tells CocoBase if a select, selectblock, selectNext or selectAll was called by the client, and the software can respond accordingly.

```
public static int CB_SELECTONE=0;
public static int CB_SELECTBLOCK=1;
public static int CB_SELECTNEXT=2;
public static int CB_SELECTALL=3;
```

Since a callback may be shared information, CocoBase needs a way to determine the 'context' of which user and which connection the callback is dealing with. In each of the methods, an instance of the thought.CocoBase.CocoUser is passed into the plug-in. This is the context Object, which contains all of the information of the user, its underlying mapping connection and any other private data that the plug-in wishes to store in the Object. Each of the CocoUser instances has the following public 'get' interfaces (note that each has a corresponding set operation available):

```
public class CocoUser {
    public String getCocoConnection( );
    public String getCocoDriver( );
    public String getCocoURL( ) {
    public Object getDBConnection( ) {
    public String getPassword( ) {
    public String getUser( ) {
    public Object getUserInfo( ) {
}
```

The getUserInfo( ) method (and corresponding setUserInfo( ) method) is used by CocoBase to retrieve a plug-specific Object. This can be any object (include lists of other objects) that will prove useful in the plug-in. This provides a very simple and flexible system that is fully extensible, and still easy to program.

There are several examples of plug-ins included with the published CocoBase evaluation distribution, and analyzing the working programming source code included as part of the distribution is often the best teaching mechanism. The caching mechanism makes extensive use of many of the features described above.

Client Programming Applications Can Access Multiple Server Databases at Once

The mapping server provides a convenient platform to access and map and enterprise set of data sources, not just a single database. This provides a convenient system for porting data from one data source to another data source (see above).

Below are several examples of how a 3-tier client using CocoBase can access multiple databases from a single client object programming application:

1) The client can open up multiple connections to the same (or different) CocoBase Server—each having it's own database parameters and logins.
2) The server connection can access multiple physical databases through a software product called a 'Gateway'. These systems create a virtual relational database over top of a user's physical databases. Vendors such as Computer Associates, Oracle and B2Systems all have Gateway products that can be accessed from JDBC drivers, and such JDBC drivers can easily be connected to CocoBase.
3) If both sources being accessed are Oracle, Oracle DBLinks can be used to link in tables from other instances into the current instance being accessed.
4) Multiple physical connections can be accessed by using a plug-in which is easily obtained by extending one of the sample plug-ins shipped with the CocoBase evaluation distribution along the lines described above and in the instructions of the distribution, and loading the extended plug-in into the Mapping server.

A simple example for extending one of the shipped plug-in would be to have two open connections, one connection to a first database and a one connection to a second reference database. When the client wishes to first try to obtain data from the second database (for example, the second database is a faster or smaller memory resident database), a shared connection to the second database is left open in the plug-in, which would append a list of Objects available from that database to the getcolumns lookup. The connection to the second database would integrate into the selected plug-in to provide a pre-process state so that data requested from the first database by the client program will cause the CocoBase runtime libraries to first look in the second reference database for an object that is present in the reference list of available Objects before attempting to access the first database that also has an open connection.

There are other ways for modifying the plug-in to accomplish the same result. For example, a plug-in could be modified to use the CocoUser storage area to store additional Connections or lists of connections to be accessed first in response to efforts by an application client to look up Objects.

Object Caching for Performance Enhancement

Object caching is provided by reading an Object (usually representing an underlying database row or fetched result set) instance and placing the data in memory for future reference while causing the database connection of CocoBase to remembers that the data is now in memory for future reference and it is unnecessary to access the database for that data. Using this technique, if the same Object instance is requested more than once by a programming application, it can be simply pulled from memory, instead of having to be retrieved from the physical database and the related overhead associated from that operation. Since subsequent reads are read from the in-memory copy instead of the underlying data storage system, this technique can be VERY efficient and highly enhance performance. If the database server is

over burdened, this can become essential for optimizing and increasing overall system performance. Since the Object Cache allows users to retrieve data quickly and efficiently from memory, it can often avoid the need for object look-up again, even with multiple clients.

With CocoBase Object Caching, as the data for each object (or the object in the case of accessing an object database) is loaded from the data source, the loaded object is stored in an in memory object cache. The Object must be uniquely identifiable to be cached-in other words a 'unique' primary key field or composite key fields must be defined. This cache is shared by all of the clients accessing the Mapping Server, and, as a result, only one copy needs to be in memory at once.

The CocoBase Object Cache contains a method for looking up the object in terms of its unique object identifier column(s). Before an object is loaded from the database by the CocoBase runtime repository, it is looked up in the cache by the identifier column(s). Using this technique, an object present in the cache is loaded from the cache instead of requiring the system to access the data source and create a new instance of the object. If the object is found in the cache, then that object is returned from the cache. A developer could create the cache in a way that would cause it to grow and fill all available memory, so it is wise to use a large memory size for the Java startup parameters. If a VERY large set of tables needs to be cached, then something like the Gemstone/J Java virtual machine should be used. Their virtual machine can grow with its own virtual memory, allowing the cache to grow as needed.

Caching with the CocoBase Enterprise Optional Mapping Server

CocoBase Enterprise provides two separate examples of a shared central cache source code examples of optional mapping server plug-ins. These programs are CocoServerSampleCache.java and CocoServerSampleCachePool.java, and each is included as both source and pre-compile binary forms. The plug-ins can be modified for custom caching, or they can used as they are shipped to provide useful out-of-the-box shared caching. The shipped compiled binary code classes belong to the thought. CocoBase package by default, and should be copied into the subdirectory . . \classes\thought\CocoBase if the developer finds it necessary to modify the source code and then re-compile the modified code to provide new pre-compiled binary forms.

In the demos directory of the CocoBase package there is provided a cache configuration file, which can be used to configure Map keys that will be used by the system for caching features. The file CocoKeyBundle.properties found in the demos directory of the installed CocoBase software package, can be edited to customize caching easily. Just add entries for each Map which describe the key or keys of that map, which will comprise a unique 'key' utilized in identifying each unique instance. The format is 'Map=COL[, COL2[, .]. .]'. To provide an Index for the Map 'Customer' using both attributes 'name' and 'address' as the composite unique key, place an entry in the CocoKeyBundle.properties file which has the following format:

Customer=NAME,ADDRESS

Note in the above entry that the attributes are ALWAYS upper case, and that they match the 'virtual' column name used in the map (for example: NAME and :ADDRESS even if columns are NAM and ADDR).

The client (presuming they have permission through proper assignment of their admin ids by system administrators) can "additional override" this default value, but using a CocoKeyBundle.properties file will allow the caching to be configured on the server without having to code or make client specific changes. By using this default properties file, each column that is included in the properties file is made part of the lookup key by the cache.

In some cases, it may be desirable to allow the client to configure a cache, and the following example code accomplishes the same task. This example code is extracted from the demo program CocoDemoCache.java included in the demos directory of the installed CocoBase software package. The example code uses the CBSETCACHEKEY "virtual Object" and "procedure call", both of which can be intercepted in the caching plug-in, and the plug-in can be set to over-ride each of them to manage caching configuration.

```
//
// Set up a searchable cache key for the Customer Map.
// Since the demo database doesn't have primary keys supported
// in JDBC, we can simulate a primary key by configuring.
//
Vector cbkeyfields = myBase.getColumns("CBSETCACHEKEY");
    if(cbkeyfields != null) {
        System.out.println("Server is cache Configurable,
configuring Customer object cache.");
        // Configure Key information for Customer so server caches
it
        // properly. We only have to do this because
        // SimpleText doesn't support primary Key fields. If it
        // did, this wouldn't be
        // necessary.
        GenPObject gpo = new GenPObject( );
        Properties props = new Properties( );
        // We want to index on the NAME attribute
        props.put("KEYLIST","NAME,ADDRESS");
        props.put("OBJNAME","Customer");
        props.put("CBADMINUSER","cbadmin");
        props.put("CBADMINPASS","cbadmin");
        gpo.setPropObjectData(props);
        myBase.call(gpo,"CBSETCACHEKEY",0);
}
else
        System.out.println("Server either doesn't support Caching
or isn't cache configurable.");
```

While it is much simpler to configure caching on the server through the properties file (and preferable) as described above, client side configuration is also possible using the technique described above. Both plug-ins keep the cached objects in a local in memory server cache until the changes have been committed into the database. Once the changes to objects are committed into the database, the cached objects are merged into the central shared cache and are then accessible by other clients needing those objects.

Performance can be enhanced by maintaining a list of previously instantiated objects in an "in memory" cache list of objects in the Optional Mapping Server. This can be readily accomplished through using a variety of data structures and facilities including Hashtables, Vectors and Object databases such as Gemstone (which use a technique called "persistence by reachability") to extend the Java Virtual machine to have Virtual Memory.

System Performance Increases are Possible with Caching and CocoBase

CocoBase can support several thousand direct database requests per minute even on a simple Pentium server with garden-variety databases. Caching provides an even better scaling promise, especially for heavily accessed databases. Services such as Caching and connection pool management can significantly enhance performance numbers. Tests of the caching methods when CocoBase was being improved to implement the above concepts have shown that performance ca nearly triple through the use of even simple caching

techniques. For example, some internal benchmarks with CocoBase and caching have clocked a 233 MHz K6/Pentium processor machine with Windows NT and 64 MB RAM as being able to serve up about 20,000 database requests per minute for individually requested cached data objects. The 20,000 requests did not constitute high volumes of data (about 30 bytes per request—a simple Object), but nonetheless such benchmarks can give developers a "rule of thumb" idea of potential increases in throughput due to caching, even when a very low-end and non-tuned system is used to serve up responses to database requests.

Other tests with CocoBase and caching used a small matrix test suite to show approximately how many cached read operations per minute a developer might be able to expect on a sample Platform. The numbers also indicated how accessing blocks of data can actually increase the overall network throughput of objects that a server is capable of handling. For example, blocking data in access blocks of 100 instead of access blocks of one can actually increase the server throughput to 5 times the data volume even when though the number of actual requests, which could be served, was reduced. These tests might encourage developers to use the block API wherever possible, and then tuning the access block sizes to an appropriate size for an individual application to enhance performance.

In some caching tests, a block size of 20 had a higher throughput overall than a block size of 100, a fact that may be related to the mechanics of how CORBA handles large blocks of I/O.

The above performance numbers might be merely considered guidelines, since performance numbers such as this can be very network, ORB and data specific. However, these test do clearly demonstrate that using CocoBase with caching (Mapping Server Cache) in combination with the tuning of block size can have an significant impact on performance, and can sometimes provide a significant throughput increase for application/database servers.

There are obviously additional ways to tune specific environments and configurations even outside of CocoBase, since the above numbers are mainly given as approximations that demonstrate the CocoBase system and technology that can give developers real world examples of tuning performance, and perhaps a ballpark idea of what might be possible. Performance enhancement of an individual system and application configuration may vary based on Java Runtime, OS, Database(s) used, networking configuration including and bandwidth capacity, ORB, etc.

Where caching really can really be beneficial to a system is in the client's 'perceived' performance, and when 'blocks' of data can be successfully cached (i.e., the application does not need to wait for an SQL statement to execute in order to access and use data for processing a request, and the system does not have to wait for the same unchanged records to be accumulated in memory over and over again), a system can seem faster because of the reduced processing delays. The CocoBase system uses caches that are written to serve multiple users out of a single pool, and by default the CocoBase cache system is a write-through cache for extremely scalable performance numbers.

Such a write through cache is written for a CocoBase system designed to provide scalable performance while avoiding certain undesirable impacts on the transactional capabilities of individual application servers. Since caching is a complex science, it should not be tackled at the beginning of a project that uses the CocoBase system. Instead caching should be implemented and fine-tuned as a final step in development. Such techniques can allow a developer to refine and improve an already streamlined system and avoids using a cache during development that could lead to unfortunate masking of poor software designs and system design flaws. Such problems might only become evident when the final application is implemented and scaled, if performance issues arise, which were masked by the caching during development stages.

Enabling Optional Mapping Server Object Caching

In the Optional Mapping Server that can be used for object caching, caching is implemented in the Server Plug-ins instead of being within the Optional Mapping Server itself. This can provide more flexible caching that is user extensible. If either the CocoServerSampleCache or CocoServerSampleCachePool plug-in is loaded, all objects having a database primary key that is accessible from a JDBC will be cached by the CocoBase system. In this system, object data caching is done through the Optional Mapping Server plug-in APIs. Source code jump-starts for providing caching are included with the Optional Mapping Server as source code, but the CocoBase system does not automatically integrate caching directly into the Optional Mapping Server. Such integrations should be saved until the later stages of project development to avoid masking design flaws.

At the final stages of development, a developer can utilize the source code provided with the CocoBase package and implement a specific kind of caching necessary for the design of the developed system, and integrate that caching system into the Mapping Server. As described above, when the source code of the examples provided don't provide explicit functionality desired by a developer, the code is extensible and can be modified to meet specific project requirements.

Sample cache provided can yield many of the common features needed by developers when the CocoBase system is implemented, and the sample cache will probably be completely useful without modification for many tasks. Modification and implementation of caches are an advanced developer functionality issue. The steps of implementing and tuning caching should only be taken on as a final step before deployment, and be taken on with careful planning. Blindly implementing caching without fully understanding a system design and understanding the purposes for caching in the context of that system design can create serious performance, memory and data integrity issues. For example, having multiple databases open in combination with caching can raise real issues with respect to data integrity if caching and multiple databases are not managed properly, particularly where database accesses are not properly managed and synchronized.

Managing Object Integrity for Applications Using a CocoBase System

Object integrity is typically maintained by keeping the cache synchronized with the data in the underlying data store, and by ensuring that any changes are reflected in the cache if, and ONLY if, such changes are successfully committed to the database. When a programming application ends a transaction with a commit operation, any updated persistent objects are written back to the data store, locks are released, and the client specific cache is invalidated (this is referred to as integrating the client specific cache into the shared cache). When the application ends a transaction with a rollback operation, updates are discarded, locks are released, and the client specific cache is invalidated. This will leave the shared cache unaffected and unmodified.

For some application environments, objects may additionally need to be retained in the cache across transaction boundaries (in other words it may be desirable to integrate

a change into the shared cache before it is committed). In this case, some application-specific logic is needed to deal with the need for consistency between the objects and the underlying data store.

Another important issue associated with Object Integrity is the need for unique Object Instances. If duplicates are allowed, it can become nearly impossible for a system to cache and to track Object Instances without the use of the transactional capabilities of a server to synchronize the cache and the main datasource. Because of this, some 'composite' key may be required to ensure that unique instances exist. If a unique key definition exists, then the Mapping Server can manage Object Integrity more properly.

Differences Between Map Caching and Data Object Caching

Object Caching and Map Caching should not be confused. Object Caching caches the user data, and must be implemented in a Optional Mapping Server plug-in. Map Caching automatically occurs in a CocoBase connection. Since Maps can become quite complex depending on the number of where conditions and fields involved in its definition, it becomes critical to Cache connections by default. As a default, CocoBase meta data for each Map is cached by the CocoBase system the first time that the Object is accessed.

The meta data caching can be triggered for a particular map before the data from the map is utilized by calling the method getcolumns( ) on the Connection with the Map name as the parameter such as:

```
Vector tempVect = myBase.getColumns ("Customer");
```

In this example, a Customer Map will be accessed and cached in the CocoBase runtime before any SQL is created or allowed to execute. This is a convenient method designed to speed runtime access of Maps in the Mapping Server through pre-loading and caching the Map. If this pre-loading and caching is necessary for Maps that are likely to be accessed by clients, it can be done at server/connection startup in order to improve perceived user performance characteristics. Once the server/connection is achieved the system is ready for database accesses, there is nos second delay required to pre-load and to cache the maps before accessing data.

Because some systems may have literally thousands of Maps for a single connection, the CocoBase runtime is not configured by default to auto-load all maps, only those specifically requested to be loaded or accessed directly will be automatically loaded. In the case of Map caching, a lazy cache is more effective. Especially since with a lazy cache it is so easy to trigger a load and cache of particular Maps that are likely to be used.

CocoBase is currently configured to only permit an application to dump a Map cache from a connection by closing and re-open a connection. Such a dump involves only two method calls, and is easy to implement if necessary by simply entering calls similar to the calls to the "MyBase" database that following two lines illustrate:

```
MyBase.close( );
MyBase.connect( );
```

These two lines will cause all of the Meta Data Map information related to this database to be reset from the connection.

Object Uniqueness Requirements in a Cache

Since it is possible with most databases to have duplicate records in a table that maps the data structure, this can prove problematic for providing a caching function that can server up unique Object Instances. The CocoBase system and runtime resolves this issue by using a default setting with respect 'unique' keys. This default setting will only permit caching of those records that are specified by a 'unique' key, which identified through the Database Meta Data available from JDBC. Additionally, it is possible for a developer to define a unique key column (or set of columns) through a startup init file or through the runtime API. Only a developer with advanced system design understanding and high level programming skills should attempt to make such a modification, since this is a task that should only be done if instances to be stored in the cache through the map are truly unique.

The CocoBase system is designed to manage the cache by using a unique key, or the key column(s), as the unique identifier in looking up instances and then stores the instances in a Java Properties Object with the unique key as the identifier. With this design, if two duplicate records are retrieved from the database, only one instance will be preserved in the Properties based cache. The CocoBase is not designed to create uniqueness or to enforce uniqueness for database tables that do not have unique index defined. Because of this, the CocoBase system is designed to ignore an attempt to create caching or to use caching if a unique primary key has not been defined in the database, or a unique key is over-ridden by the developer by their software application design.

The CocoBase system is designed to allow non-unique objects to be accessed just as in ordinary database accesses where the CocoBase system is not used. However, the use of as caching is not turned on for those Objects. If a database table doesn't define a unique key, the Objects simply get passed through the CocoBase plug-in without caching becoming involved. However, a developer can easily take advantage of CocoBase plug-in caching even if a database table does not use a 'unique' key column set, by simply defining their own unique key for the database table.

Developers can look at the CocoKeyBundle.properties startup init file to see how the CocoBase system provides an ability to dynamically define a Unique Key. Also, developers can review the CocoDemoCache.java file portions related to the CocoBase connect ( ) methods format and commands, for an example of how a developer can easily define an Object Unique key through the use of the Mapping server CBSETCACHEKEY functionality.

Design of the CocoBase Sample Optional Mapping Server Object Caching

As described above in several text locations, there are two sample Caching plug-ins included in the CocoBase system distribution and the design of each plug-in is bid different. One plug-in is a "Cache only" plug-in with the file name CocoServerSampleCache.java, but the other plug-in named CocoServerSampleCache.java is designed to provide a combined caching and connection pooling.

The cache portion for each plug-in is basically identical to one another, and the cache design is fairly intelligent with respect to caching systems. The cache portion of the plug-in keeps a rollback log for each client that is using the cache (a private cache) which is integrated with the shared cache upon the completion of a successful commit that updates the datasource. If the cache setting of AutoCommit is used, the caching system of the plug-in is designed to automatically integrate the changes to the private cache into the shared cache after each operation is completed that changes the private cache.

The CocoServerSampleCache plug-in is loaded as the default plug-in for the script startcache.{bat/sh}. The other plug-in is used by substituting the script startcachepool.{bat/sh} and CocoServerSampleCachePool is loaded as the CocoBase system plug-in instead of the default cache plug-in.

The caching portion of the plug-in is also designed to support some administrator control functions for starting, stop, and managing the cache. Additional features may be added by modifying the source code that is provided as part of the CocoBase system. The cache plug-in source code described herein (and to be released after the filing of this document) supports (among other things) the administrator control functions of CBSTARTCACHE, CBSTOPCACHE and CBSETCACHEKEY, which can be used by a developer to set a caching system design that will start and stop the shared cache, as well providing the ability to override the primary key column(s) used at a resource to lookup Objects in the cache.

The cache plug-in source code does not need to be modified to manage transactional coordination or regulate cache memory size management. Cache system design is easily configured to solve such issues in a much more reliable fashion. Cache databases, for example, ODI and HYPERSONIC SQL each implement both manual and automatic mechanisms to trim the cache size on low memory availability. Other cache management features are provided by server monitoring capabilities, such as the multiple database synchronization capabilities of Weblogic 6.1, as described earlier. If the cache database and the main database are both registered with the Weblogic 6.1 server (or other application servers having similar monitoring capabilities), the transactional capabilities of the server can be harnessed to manage synchronization of the cache database with the primary database in coordination with the CocoBase system and fully implement the cache plug-in features of CocoBase for enhanced system performance.

Optional Mapping Server Definitions and CocoBase System Integration

The CocoBase Optional Mapping Server is designed to integrate into and interoperate with standard Application Servers whose users need to have database access. The CocoBase runtime and Optional Mapping Server can be used to provide persistence and mapping functionality to the overall system irrespective of the type of object programming application server that is chosen to be utilized for a particular computer system.

Generally speaking, application servers enable corporations to develop, deploy, and manage business-critical applications that frequently need to have transactional coordination capabilities. Object programming applications that run under such servers can delegate certain functions, such as the transactional coordination to the server.

Many sound reasons exist for developing applications that can run under such servers and can utilize the CocoBase database access features for development and deployment. A few of those reasons are high performance, rapid development, scalability, easy integration with existing enterprise systems, state management, load balancing, and table/row locking mechanisms. Such features can be the keys to good html and Jave Server Pages (JSP) creation that can interface well with a user and permit the user to access to data and to conduct transactions that implement important features of the server system and the CocoBase system that are invisible to the user. Examples are the server transactional coordination and the CocoBase runtime and caching systems. Such software application programs can be developed to delegate the functions of database access and data persistence to the CocoBase system, without the need to include database specific or persistence specific code in the application program source code. Moreover, CocoBase can automatically generate the server connectivity source code and persistence source code from the database maps.

CocoBase leverages the functionality of the Optional Mapping Server and features of application servers to provide additional advance mapping functions—typically the weakest link in Application Servers. CocoBase has become the pre-eminent mapping technology in Java, and provides superior mapping that can be readily utilized with virtually any mapping server as part of the overall system design. Application servers should be thought of as very complimentary technology to the CocoBase database mapping features, the CocoBase mapping server, CocoBase cache plug-in features, CocoBase runtime SQL generation facilities as well as other functionalities of the CocoBase software system product line.

Connection Pooling Systems with the Optional CocoBase Mapping Server

Connection pooling may be described as the concept of keeping database connections open and re-using them with new clients. Since it can be a resource intensive prospect to open a new database connection it is often advantageous to reuse existing database connections when possible, and to even use cached objects that are obtained from an earlier access of the database. A connection pool can be created for multiple clients. It keeps the connection to the database open for future requests. Creating a pool of JDBC connections gives clients ready access to connections that are already open. Connection pooling can improve performance by removing the overhead of opening and closing a new connection for each database operation or each time a new user tries to access the same database. The connection management service of CocoBase can determine whether a database query can use a connection from the connection pool based on data source, user name, password, and the logged-on user account, etc. The connections in the pool can be shared among any users who can successfully match the login requirements. With this system design, each member client can get a connection from the pool for use, and then return it to the pool when finished (but the connection itself is not closed).

To configure the connection pooling of the CocoBase system, the developer can simply edit the Pool configuration file to set the desire connection pooling requirements (the file CocoPoolConfig.properties). Both the CocoServerSampleConnectionPool.java example plug-in and the CocoServerSampleCachePool.java example plug-in are designed to utilize this property file in order to configure the system Connection Pooling. The source code of each Connection Pooling plug-in can be readily customized to meet any project requirement not covered automatically in view of the discussion and description provided herein.

CocoBase System Optional Mapping Server Overview

CocoBase provides a solid infrastructure to allow easy user plug-ins for state management, load balancing, caching, connection pooling, etc. Because the focus of this server is on mapping, it can be referred to as an Optional Mapping Server. Such a mapping server it is designed to tie together different data base driver systems, and provide transparent and consistent use of those systems for an application developer. Another good use of such a system it to permit the easy transport of data from one data system to another, as described above in detail.

In one implementation, the CocoBase Optional Mapping Server is designed as an ORB Service loaded into the

CORBA ORB. In such a case, each mapping server is fully multi-threaded and capable of supporting many simultaneous users. The CocoBase Optional Mapping Server also provides access to and transparent use of JDBC and either CORBA or RM1 with 3-tier CocoBase adapters to the Mapping Server, without requiring application developers to understand or to program in either of those driver systems. The CocoBase System also provides applications with an added level of security that is not possible from Internet enabled JDBC drivers.

The CocoBase system runtime, data caching, object caching, and mapping features can be readily utilized by a developer to achieve a very scalable and high-performance data access and mapping system. This is accomplished by minimizing I/O between the client and server, in a fully multi-thread multi server architecture. The developer can permit client applications to decide when and where and how much data is sent between the client and server. This means the system does not need to burden down data pipes with irrelevant information or with lots of small independent network requests. Because less I/O is used, a system can be designed which utilize the same data bandwidth to support many more users with the CocoBase system implemented than it could have supported in the absence of CocoBase.

Optional Mapping Server Security with CocoBase System

Object programming applications can delegate database accesses to CocoBase runtime and mapping repositories to create increased security on computer systems. This can be particularly helpful when the system is connected to the internet or other open connections. Applications that interface with uses are more secure if they have been written to utilized the CocoBase system and to delegate the database accesses to CocoBase. In such a case, clients (system users and application software) access the optional mapping server through a CocoBase map. Each map explicitly specifies each table field and where conditions associated with available operations, and the client does not have access to the underlying database driver. This can be utilized to restrict the data that can be access by a particular client. The CocoBase System provides this additional security because database access is done through the list of maps available to a particular user, and the client or application software does not have direct access to the underlying database driver and therefore to data other than that available through the list of maps.

The CocoBase system is designed to provide this additional layer of remote security by providing a list of maps to which a particular user is granted access. Even this security feature can be easily extended by customizing the source code of the Optional Mapping Server Plug-in.

The CocoBase Optional Mapping Server model does not place SQL in client applications (which reduces rogue client capabilities). Instead, the runtime libraries of the CocoBase system as described at length earlier above. The CocoBase runtime libraries in combination with the pre-defined maps manage all SQL access to the data source through the Mapping Server and predefined Maps. An implication of this design means is that if a given CocoBase map, an operation within a map, or an attribute within an operation isn't published to the client, it does not exist from the perspective of the client.

By contrast, the CocoBase Relational 2-tier system sits directly on the JDBC driver instead of utilizing the mapping server and the runtime maps, and the 2-tier system does not possess the same security mechanism. In such a system, the CocoBase 2-tier runtime libraries permit a getconnection ( ) call, which returns information necessary to access the system JDBC driver. As a result, it's not possible to provide additional security over JDBC if such a 2-tier system is employed. Many system that are not truly an "O/R" system in that they are not fully delegated the control of database assess have the same problem. In such a system, security will be limited to whatever security is provided by the JDBC driver and the associated database, such as database login protocols and the like. Such login protocols can be limited. Typically, if a database user has the right to can log into a database, then all of the data in that database is accessible to that user. This can require a system design of multiple databases where part of the data is duplicated, which can be hard to keep synchronized and uses additional storage resources.

In view of these additional security features due to the CocoBase system design, developer that need to provide access over the Internet should use a 3-tier CocoBase adapter and the associated mapping server. The CocoBase Relational 2-tier system and other similar systems may be sufficient mainly for Enterprise 2-tier Intranet application development. However, even in such systems it may be desirable to have a single database where certain data needs to be secure from access by some system users. The Optional Mapping Server provides applications an added level of security not possible from Internet or Intranet enabled JDBC drivers. Due to the security design flaws present with typical JDBC drivers, and the concept of permitting a JDBC driver is to pass arbitrary SQL from client applications to back-end server systems, it can be difficult or virtually impossible to secure JDBC systems for Internet or Intranet use without the use of the CocoBase 3-tier system of delegated database access.

The CocoBase system may also be described as having architecturally secure design that works on the 'publish paradigm' with the server managing and publishing all access to SQL on the database server. Client access is restricted to only those defined maps, which are published as being available from the Mapping Server. With this Map focused design, the CocoBase Enterprise 3-tier system (with its delegated database access features) allows companies to pre-define maps that can be executed by clients, instead of allowing clients to define or change the SQL statement that the server is capable of executing. Through adapting the CocoBase plug-in source code, developers can easily integrate/extend the components for providing additional security, and to delegate security and project specific logic to the CocoBase system.

Maps and objects related to those maps can be created in a number of ways, usually in response to application users requests. Accordingly, the system of the present invention will have to accommodate such numbers of objects and maps associated with those objects. Because of the flexibility of the present system, maps can be bonded to objects dynamically (during the application run time). The result will be that a single map could be appropriate for a plurality of objects. On the other hand, a single object may be bonded by a plurality of maps. Objects that can be used to access large volumes of data, the use of any particular map may initiate a data response so large that it cannot be easily managed. Accordingly, single object may have a number of different maps each configured to access only a particular portion of the data associated with the selected object. In this manner, maps can be used to select only that data which would be relevant to the map user. This selectivity will render a large volume of data associated with a particular object to be more manageable for access and use. Likewise, the distributed of selected maps to access only selected

portions of the datastore associated with a particular object can be used in as a security device. In this manner, individuals are provided with maps having the capability of accessing only those portions of the database that are appropriate for the map user. Thus, the selection and distribution of maps for a particular object can serve as a security hierarchy with respect to the data to be made available.

The dynamic mapping of the present invention can be applied where a Java object provides translation by mapping such an object to first data source (either relational or object) can also by mapping such an object to an XML for a second format database. This arrangement allows greater control as to how the data steps are exchanged, filtered, and validated by exchanging data.

All references documents, including software files accessed via internet, are incorporated herein by reference. Since these documents are already published, they will be provided under Rule 56.

Light-faced carrier type is used to denote command names, subroutines, files names, software modules and other variables.

While a number of preferred embodiments of the present invention have been discussed by way of example, the present invention is not to be limited thereby. Rather, the present invention is to be construed as including any and all variations, modifications, permutations, adaptations and embodiments that would occur to one skilled in this art once having been taught the present invention.

I claim:

1. A computer system comprising at least one computer implemented program, at least one data source, and a mapping system wherein object programming applications are tailored to delegate both the accessing of a data source arid the generation of SQL strings to a runtime library repository wherein the repository comprises a collection of system information and software program libraries, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanism classes in the application code of an object programming application prior to runtime, wherein:

i) the mapping system portion is designed to provide to a system user who is accessing, creating or updating maps, or accessing objects on a system to make data changes related to a particular object and to promulgate the changes to that object as either local or global changes on the computer system, or

ii) the mapping system portion provides an interface permitting an authorized user to edit or create the tables, fields, or attributes of a data map for an object as a table format or XML file format without requiring the user to have extensive knowledge of a particular relational database as a source of the data, or extensive knowledge about how to directly access that relational database.

2. A system according to claim **1**, having a concurrent parameter setting mechanism wherein the runtime library repository can be set to access a particular data source and to generate data source specific database calls and SQL strings.

3. A object language software program that can be implemented by a computer to generate application programming code from stored database maps and thereby provide a programming application without the need for database accessing mechanism classes within the application itself by including with the application classes object code logic that delegates to an independent and modifiable runtime library repository both the functions of accessing a database and generating SQL strings that are specific to a database or specific to a JDBC driver for a type of database, wherein the software logic comprises:

i) a software module providing logic for a system user who is accessing, creating or updating maps, or accessing objects on a system to make data changes related to a particular object and to promulgate the changes to that object as either local or global changes on the computer system, or

ii) a software module providing logic for an interface permitting an authorized user to edit or create the tables, fields, or attributes of a data map for an object as a table format or XML file format without requiring the user to have extensive knowledge of a particular relational database as a source of the data, or extensive knowledge about how to directly access that relational database.

4. An object language software program according to claim **3**, wherein the runtime library repository can be modified and tailored to optimize database access calls and to optimize the generation of SQL strings for a particular database, without requiring any modification to the logic of object classes in applications that will delegate database access functions to the repository beyond the inclusion of logic instructing the application to delegate database access functions to the runtime repository.

5. A computer system according to claim **1**, wherein the mapping system portion is designed to provide different maps for particular objects to different users and does not permit direct access of computer system users to JDBC drivers for any mapped data sources of the computer system, and wherein this mapping system provides varying levels of access to the mapped data sources for at least two different users of the same system, whereby a user only has access to a particular list of maps that are available to the security level of that user.

6. A computer system according to claim **1**, wherein the mapping system provides an interface and features that permit a user to access, create, or up date the metadata of a map as a dynamic computer system update, without requiring the user to either open a new connection to the data source or to restart an object application program that is running while the user is dynamically evaluating or changing metadata for a map, and wherein the metadata of a map that a user can dynamically evaluate or change includes a map description of data or relationships between data, and wherein such map description is at least one member selected from the group consisting of a data source relationship, a relationship between at least two objects of an object application, and both a data source relationship and a relationship between at least two objects of an object application.

7. A local or distributed computer system comprising fully synchronized caching system utilizing at least one computer implemented program that synchronizes the caching of a delegated data source access management object to relational mapping layer and the transaction coordination facilities of an application server, and the computer system comprises (i) a first data source referred to as the primary data source, (ii) a second data source referred to as the cache data source that is associated with an object to relational mapping layer to provide a data source cache for object applications, and (iii) a server having a transaction coordinator with the ability to register multiple data sources, wherein:

i. both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and

ii. the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and

iii. registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources, and the computer system further comprises a mapping system wherein object programming applications are tailored to delegate both the accessing of a data source and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanism classes in the application code, and wherein the transaction monitor of the server calls the mapping system and in the call delegates accessing or updating of data in the data sources registered with the transaction monitor, and wherein the transaction monitor does not complete a transaction until it is notified by the mapping system that a data source has been updated by the mapping system such that any changes to the cache will automatically be synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

8. The computer system of claim **7**, wherein at least one cache data source can be set up as a memory-resident entire database, as a disk resident database, or as both a memory resident database and disk resident and in each case the cache data source is synchronized with the transaction monitor of the server.

9. The computer system of claim **8**, wherein the at least one cache data source is a memory resident database that is a portion of a disk resident database and the size of the memory allocated for the memory resident database can be set by the user of the computer system, or can be automatically allocated by the system according to pre-set protocols.

10. The computer system of claim **9**, wherein the at least one cache data source is adapted to cache at least one member selected from the group consisting of data from a data source, a source structure map, an object or object definition, metadata, object schema, an object schema, and an XML file which defines or describes a member selected from the group consisting of data from a data source, a source structure map, an object or object definition, metadata, and an object schema.

11. A data source access repository capable of handling delegated database access functions when said repository is accessed or utilized by a computer implemented program, wherein the data source access repository is a collection of information repositories and software program libraries and comprises at least one database access runtime library repository of software program libraries and system information wherein:

(a) the data source access repository has the ability to make database specific calls and to generate database specific SQL strings by accessing at least one other software program library, information and logic to permit the data source access repository to map information related to one or more of a member selected from the group consisting of relational databases, object databases and object programming schemas, and the like,

(b) the data source access repository can access map information and use that information to make database specific calls and to generate database specific SQL strings that can be passed to a JDBC driver to access, retrieve and write data to and from a selected data source, and

(c) a table, field, or attribute of at least one data map for an object, when that at least one map is included within the runtime repository, can be accessed and edited by software logic in a table format or be edited as an XML file format without requiring the user to have extensive knowledge of a particular relational database as a source of the data, or to have extensive knowledge about how to directly access that relational database.

12. A data source access repository according to claim **11**, comprising one or more items selected from the group consisting of imps of database schemas, object definitions, other programming definitions or parameters, object metadata, database settings, and complex relationship definitions.

* * * * *

EXHIBIT D

US007043481B2

(12) **United States Patent**
Mullins et al.

(10) **Patent No.: US 7,043,481 B2**
(45) **Date of Patent: May 9, 2006**

(54) **SYSTEM, METHOD AND SOFTWARE FOR CREATING, MAINTAINING, NAVIGATING OR MANIPULATING COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS**

(75) Inventors: **Ward Mullins**, San Francisco, CA (US); **Alexandre Martins**, Floianopolis (BR)

(73) Assignee: **Thought, Inc.**, San Francisco, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 490 days.

(21) Appl. No.: **10/158,672**

(22) Filed: **May 30, 2002**

(65) **Prior Publication Data**

US 2004/0015829 A1 Jan. 22, 2004

Related U.S. Application Data

(60) Provisional application No. 60/295,012, filed on Jun. 1, 2001, provisional application No. 60/300,754, filed on Jun. 25, 2001, and provisional application No. 60/305,847, filed on Jul. 16, 2001.

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** ..... **707/10**; 707/3; 707/103 R; 707/104.1; 703/2; 709/218; 717/5; 717/107

(58) **Field of Classification Search** ..... 707/1–10, 707/100, 102, 103 R, 104.1, 200; 709/218, 709/213, 214; 103/2; 717/1–5, 107, 104; 719/328

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,107,418 A * 4/1992 Cramer et al. 717/155
5,499,371 A 3/1996 Henninger et al. 606/194
5,542,078 A 7/1996 Martel et al. 707/101
5,611,076 A * 3/1997 Durflinger et al. 707/102
5,713,014 A * 1/1998 Durflinger et al. 707/4
5,779,310 A * 7/1998 Suskey et al. 707/102
5,799,310 A * 8/1998 Anderson et al. 707/102
5,850,535 A * 12/1998 Maystrovsky et al. 703/2
5,855,013 A * 12/1998 Fisk 707/3
5,857,182 A * 1/1999 DeMichiel et al. 707/3
5,930,824 A * 7/1999 Anglin et al. 711/162
6,016,497 A * 1/2000 Suver ..... 707/103 R
6,096,095 A * 8/2000 Halstead 717/107
6,163,776 A * 12/2000 Periwal 707/4
6,215,495 B1 * 4/2001 Grantham et al. 345/419
6,243,717 B1 * 6/2001 Gordon et al. 707/203
6,405,198 B1 * 6/2002 Bitar et al. 707/6
6,411,954 B1 * 6/2002 Roffe et al. 707/8
6,662,199 B1 * 12/2003 Flight et al. 707/104.1
6,721,740 B1 * 4/2004 Skinner et al. 707/10
2002/0124055 A1 * 9/2002 Reisman 709/218

FOREIGN PATENT DOCUMENTS

EP 0 809 198 A 11/1997

OTHER PUBLICATIONS

Markowitz v M et al.: "Object queries over relational databases: Language, implementation and applications" Proceedings of the International Conference on Data Engineering. Vienna, Apr. 19–23, 1993.

* cited by examiner

Primary Examiner—Shahid Al Alam
(74) *Attorney, Agent, or Firm*—Robert Lev

(57) **ABSTRACT**

The invention provides systems, methods and software for creating, maintaining, accessing, navigating and persisting complex data objects and associated data stores. In one aspect, the invention also provides an application programming interface ("API") useful for creating, maintaining, accessing, navigating and persisting complex data objects and associated data stores as a CDOG model.

16 Claims, 2 Drawing Sheets

(Complex Data Object Graph "CDOG")

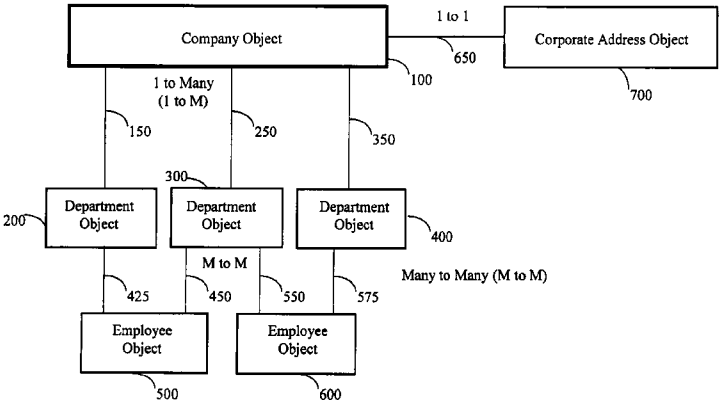
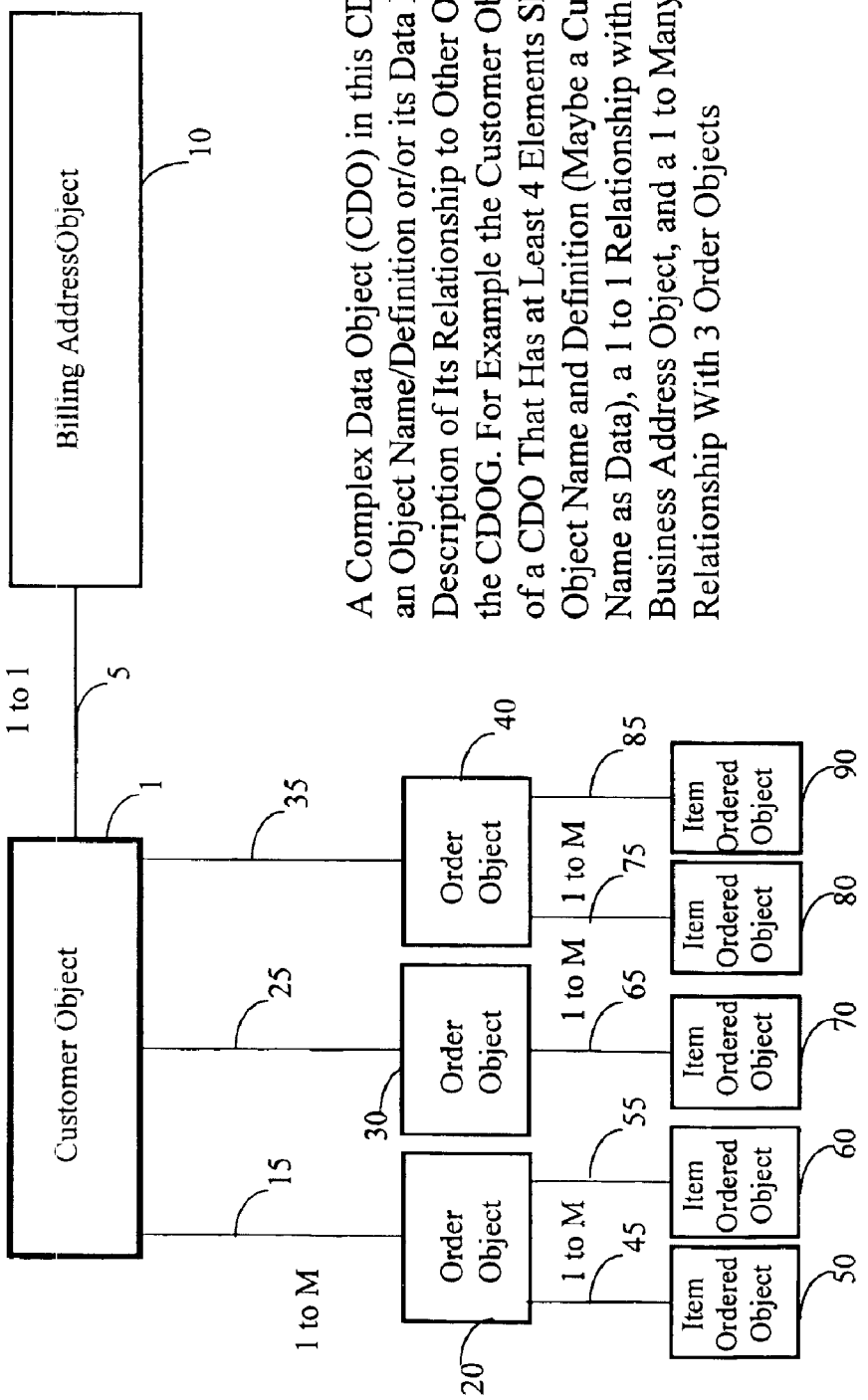
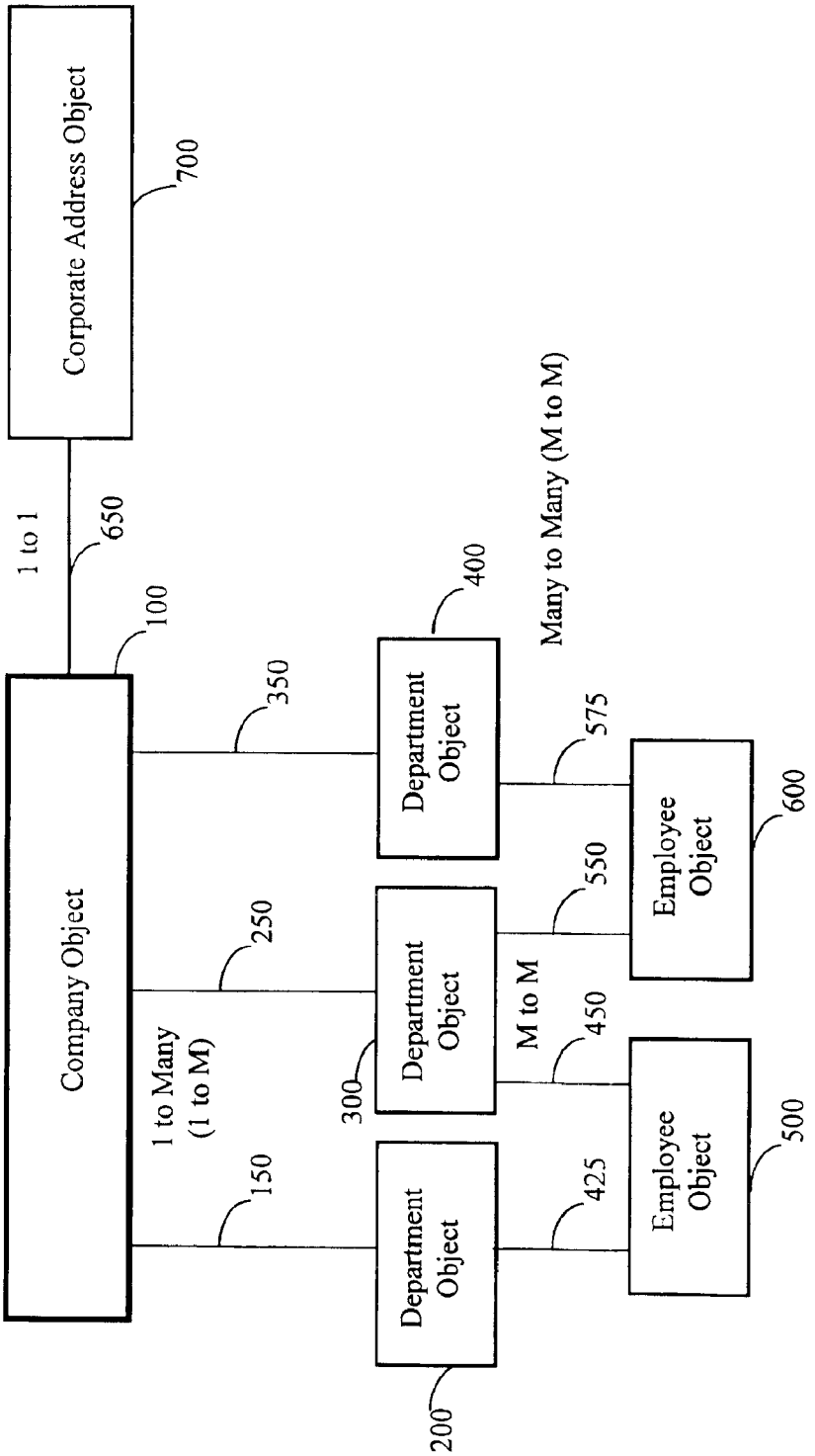


FIGURE 1
(Complex Data Object Graph, "CDOG")



A Complex Data Object (CDO) in this CDOG shows an Object Name/Definition or/or its Data Plus Plus a Description of Its Relationship to Other Objects in the CDOG. For Example the Customer Object is Part of a CDO That Has at Least 4 Elements Shown: the Object Name and Definition (Maybe a Customer Name as Data), a 1 to 1 Relationship with the Business Address Object, and a 1 to Many (1 to M) Relationship With 3 Order Objects

FIGURE 2
(Complex Data Object Graph "CDOG")



SYSTEM, METHOD AND SOFTWARE FOR CREATING, MAINTAINING, NAVIGATING OR MANIPULATING COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS

Based on Provisional Applications: 60/295,012 filed on Jun. 1, 2001; 60/300,754 filed on Jun. 25, 2001; and, 60/305,841 filed on Jul. 16, 2001.

FIELD OF THE INVENTION

The field of the present invention relates generally to computer systems, computer data stores and to methods and software for accessing and utilizing data stores. More particularly, the present invention relates to a system, methods and software for creating, maintaining, accessing, navigating and persisting complex data objects and associated data stores. In one aspect, the invention also relates to an application programming interface ("API") useful for creating, maintaining, accessing, navigating and persisting complex data objects and associated data stores as a complex data object graph, or as a set of complex data object graphs.

BACKGROUND OF THE INVENTION

Systems for accessing data stores from object oriented languages have been used for many years. A frequent approach to accomplish access of data stores involves writing and embedding custom access code within an object application needing the access. This approach is generally limited to having the custom code access only a single relational table within a relational database or similar construct within any other data store (hereinafter collectively "data store"). Under the circumstances where a developer has control over the design and creation of a data store from its inception, it is possible to design and store meaningful information in a single table. Such design opportunities are usually rare, however.

Generally, the design goals of a properly normalized relational data store or specific object data store, for example conflict with those of object application models. An object application model may exist in one or more of a computer's memory, an application data store and an application information storage repository that may be independent of the data store organization or object definitions. Whether a data source is a relational database, an object database or a flat file (text) base, procedures and systems exist for converting retrieved data to a data object, or to a data object definition, that is mapped to the data source. Advancements in the art have been made with respect to tools for conveniently mapping objects to systems of tables and maps in order to expedite accessing, changing and updating data stores. See, for example, U.S. Pat. No. 5,857,197 (and its associated programming interfaces ("APIs")) describes tools for translating object data to relational data, relational data to object data, and object data to object data to expedite the use of data stores. One or more of such data stores, data objects, data object definitions, may be utilized by an object programming application and may be illustrated as a model of data objects or as a model of data object definitions.

A further problem complicates the creation, access, changing or deleting of an object application model that utilizes such data stores. The object application model may be distributed over multiple physical computer machine locations or even distributed over multiple Internet website locations that may be independent of the data stores. The

object application model may utilize a different set of data objects or different set of definitions for relationships between data objects than that of one or more of its data sources. In most situations, the respective structures of the data sources and of the object applications model simply do not conveniently allow for mapping, accessing or changing of an overall schema of application data objects as well as any associated definitions of relationships between two or more data objects or elements within a data object.

Importantly, relationships may exist between a data object and one or more of the other data objects found in the object application model or in a data object of the data source. A relationship between one data object and another data object or with a data source may be member selected from the group of three relationship types consisting of 1 to 1 (1—1), 1 to many (1–M) or many to many (M—M). Complex combinations of these relationships may exist as a data object relationships definition for a given data object. These relationships are described or illustrated in further detail later in this document.

Objects may logically span multiple relational tables or multiple object databases, and may even be distributed over a logical (or hypothetical) computer system involving multiple physically independent computer systems or even multiple website locations. Creating, accessing, maintaining or updating an object application model can require working with multiple translation modules and require tedious and repetitive updating of multiple individual computer systems or multiple data sources in order to do useful work and keep the object application model synchronized. Such approaches are both costly and unwieldy in terms of computing and development resources, particularly with respect to Internet based electronic commerce (eCommerce) object application models.

Data objects of an object application model are often a feature of eCommerce object programming applications, where information is obtained from a data source and the data is defined as a data object (e.g., as a Java class) for use with another computer application. In practice, a data object or model of data objects may exist only in the random access memory of a computer memory system, or may be saved to either a data source or to some other type of retrievable information repository. A programmer or administrator of an object data application may not be able to exhaustively display the overall model or diagram of data objects for an object application model or to display some of its specific elements. Unfortunately, tools for accessing and displaying data objects and associated data object relationships as a complex data object graph model have not been well implemented in the field of object language programming.

A computer application can execute one or more of the following non-limiting actions with respect to one or more of the members selected from the group consisting of data, a data object, and a data object definition: access data, change data, create data, create a new relationship between one or more data objects by creating or changing at least one data object relationship definition, change or delete a relationship between one or more data objects by changing or deleting at least one data object relationship definition, access a data object relationship definition and use its parameters to access a data source or a data object, and access one or more data object relationship definitions or data objects to create a new data object or data object relationship. Any changes executed by a computer application with respect to one or more of the members selected from the group consisting of data, data object or data object definition may need to be properly persisted (permanently

stored) to preserve any changes to one or more of the members selected from the group consisting of data, a data object and a data object definition.

A data object and an associated data object relationship definition may be represented by a complex data object graph ("CDOG"). A CDOG, for the purposes of this document, may be thought of as a computer program data object graph that represents a data object having at least one relationship with at least one other data object or with itself via a circular link. When the data object of a CDOG is implemented in the Java computer program language, the CDOG may be further defined as being a Java Data Object Graph ("JDOG").

There is a need for an easily navigable tool for detecting any relationships of a data object in order to provide a CDOG definition and for persisting any changes to at least one member selected from the group consisting of a data object, any data associated with the related object, or any associated CDOG definition (i.e., an changes to the data object, data or to a relationship of the data object with another data object). For example, there is a need to be able to properly sever relationships between at least two data objects in a convenient and efficient manner by simply making a change to the CDOG definition for one of the two related data objects.

An illustration of this might be a customer account CDOG of a company where the customer has become insolvent. Using the technology presently available, a computer applications programmer or administrator would ordinarily need to individually delete or update the data in the data source, in any instances of the data object and in any instances of any associated data objects, as well as to delete or update any associated data object relationship definitions, which would be particularly tedious and repetitive in a distributed application model. There is currently no computer programmer or system administrator tool that is efficiently adapted for an object applications CDOG model with features that permit convenient and efficient accessing and deleting of a single instance of a complex data object and (without further input) results in the tool automatically deleting all existing associated data objects and relationships and the tool automatically updating all local and distributed instances of the CDOG and its associated data, data objects and data object relationships on the local or distributed computer system.

Accordingly, there is a strong need in the art for a computer applications programmer tool designed to assist a programmer or administrator in the actions of deleting, inactivating or updating of a CDOG, wherein the computer applications programmer tool can be configured to automatically delete, inactivate or update at least one member selected from the group consisting of associated data, associated data objects and associated data object relationship definitions to reflect the deleting, inactivating or updating of a CDOG. A particularly strong need exists for such a tool having the further ability to be configured to persist, propagate and reflect system wide (in a local or distributed computer system) any such changes to a CDOG instance to all instances of the CDOG and to all instances of associated data, data objects and data object relationships.

DEFINITIONS

The following non-exhaustive list of definitions is used herein to define terms that may otherwise be confusing or can sometimes have multiple meanings. Each occurrence of a defined term in the above text, in the text that follows, or in the claims of this document, is to be given the meaning ascribed to it in the list of definitions below.

"Instance" as referred to in this document in the context of computer software applications is a single occurrence of a software logical element in the memory of a computer system, such as a "class", an "object", a "data object", and the like.

"Class" as referred to in this document in the context of computer software applications is a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). In practice, a class is a logical unit used as a logical template in an object oriented language from which to allocate new instances of objects.

"Object" as used in the context of this document is a general term referring to a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). The term "object" may ordinarily be used interchangeably with the term "class" as a template or as an instance depending on the context.

"Data object" as referred to in the context of this document represents the concept of the occurrence of an object that holds data within a specific computer application domain and is likely to have its contents stored in a persistent data source of a computer system (e.g., a database server, a binary file, a text file, or even in a combination of two or more of such a persistent data sources of a computer system). A data object may exist as an independent data object without any relationship to any other data object or it may have one or more relationships with itself or with one or more other data objects.

"Complex data object" (or "CDO") as used in the context of this document refers to the occurrence of a data object that has at least one or more relationships with itself, or at least one or more relationships with one or more other data object(s). In a given instance of a CDO at least one relationship is populated as a link, as defined below. A CDO may have a multiplicity of different relationships with itself or with one or more additional CDOs.

"Relationship" or "data relationship" as used in the context of a CDO refers to the type of logical combination that occurs between a data object with itself, or refers to the type of logical combination that occurs between a data object and at least one another data object. Among other references or descriptions, such a relationship is always referred to or partially described by a "relationship type". This term is used in an object oriented language context to reference or describe any expectations, actions and limitations possible between two or more data objects.

"Relationship type" in the context of this document is a label that specifies the possible multiple combinations that can occur between a CDO and itself or with at least one other CDO. The possible relationship type labels are 1—1 (one to one), 1–M (one to many) and M—M (many to many). A given CDO may be simultaneously related to more than one other CDO through several different types of relationship.

"Link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself, between a CDO and another CDO. The occurrence of at least one populated link results in an instance of the CDO.

"Circular link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself that may be direct or indirect (e.g., linked to itself through another CDO).

"Relationship definition" or "relationship description" in the context of this document and computer software appli-

cations refers to information, or an abstraction of information, regarding a "relationship", "data relationship" "relationship type" or a "link" that can be stored, accessed, transferred, communicated, displayed or edited.

"Complex data object graph" or "CDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships.

"Java data object graph" or "JDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships that are part of a Java programming application.

"Application model" or simply "model" are essentially interchangeable terms employed herein as abstractions to logically convey a collective description or other representation for a set of complex data objects and a corresponding description or other representation of their relationships. In one respect, these terms are used logically herein provide a general way of efficiently communicating when referring to set of metadata (i.e., data about data) that describes possible data entities (e.g., objects, database tables, maps, etc,) data relationship types, and data constraints involved in a computer system or application, or in a specific instance of an application. It is important to understand the context in which the terms "application model" and "model" are used in this document. Ordinarily computer engineers refer to the "model" as an abstraction rather than a specific possibility or instance of the model as applied. However, in this document for the ease of communication abstractions of the model, possible implementations of the model and instances of the model are all referred to generally as "application model" or "model". From the context of its use the term will be clear.

"Navigation", "navigating" or "navigated" in the context of the present document refers to an action implementing at least one object to interact with a set of related objects for a certain purpose, such as creation, access, insertion, modification and deletion of an object, or of one of its relationships.

"Navigation model" as used herein is a special type of application model that is applied specifically to a description (or other representation) of how objects can relate to each other and what might be the expected behavior when a CDOG is navigated for a certain purpose.

"Object schema" is a term employed herein as an abstraction referring to the set of data object classes that describe the possible data objects that can be created, modified or maintained in an application, or describing an instance of a set of data object classes in an application.

SUMMARY OF THE INVENTION

An object of the present invention is to provide a system for creating, maintaining, accessing, navigating and persisting complex data objects as a complex data object graph (CDOG) model. In a preferred embodiment, an object of the present invention is to provide such a system that can display the CDOG model as a graph or table of complex data objects. A further object is to provide such a system is located on, or is part of, a local or distributed computer system.

An object of the present invention is to provide a method for creating, maintaining, accessing, navigating and persisting complex data objects stores as a complex data object graph. In a preferred embodiment, an object of the present invention is to provide such a method having the step of displaying or printing a representation of the CDOG model as a graph or table of complex data objects. In a further object, such a method involves a local or distributed computer system.

An object of the present invention is to provide a computer software program having at least one user interface and having at least one feature for at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, an object of the present invention is to provide software program with a module or associated software program having a feature for displaying or printing a representation of the CDOG model as a graph or table of complex data objects. In a preferred object of the invention, such a software program has an editing interface for editing a CDOG model, or an editable input or source, such as a file, that can be modified to implement changes to a CDOG model. A further object is to provide such a software program adapted for a local or distributed computer system.

An object of the present invention is to provide an application programming interface ("API"), as a preferred embodiment of the invention, wherein the API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model, or as an object schema. In one aspect the API can be accessed by a computer software program, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, an object of the present invention is to provide such an API as part of a software tool package having a component or method for displaying or printing a representation of the CDOG model as a graph or as table(s) of complex data objects. In another preferred object, such a software tool package including the API would be provided with an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the complex data object model. It is a particularly preferred object to provide such an interface as a point and click graphical user interface that may be displayed and/or edited. A further object is to provide such a API for operation on a local or distributed computer system.

A preferred object of the present invention is to provide a software tool comprising the API according (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

A further object of the present invention is to a software tool capable of reading a source programming object logic model or a database file in a format selected from the group consisting of a UML data file, m a XMI data file, and a XML file and converting the information into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file. In a preferred object, the software can automatically generate a persistence layer that corresponds to the object model information of the source file.

BRIEF DESCRIPTION OF THE DRAWINGS

For the non-limiting purpose of illustrating some of the concepts of complex data objects CDOs, i.e., data objects

and their relationships to one another, according to the invention, two CDO graph drawings FIG. **1** and FIG. **2** are provided.

FIG. **1** is a complex data object (CDO) graph drawing, which illustrates a customer object and some of its related objects (billing address, orders and items ordered), as well as relationships between the objects. Relationships of the types 1 to 1 (1—1) and 1 to many (1–M) are shown in this CDO graph. More specifically, FIG. **1** illustrates a CDO graph drawing presenting an instance of a customer object **1** having a 1 to 1 (1—1) relationship (**5**) with its customer billing address object **10**, and a 1 to many relationship (collectively **15**, **25**, and **35**) with the three outstanding order objects **20**, **30** and **40**, respectively. Order object **20** is an instance of an outstanding order object having a 1 to many relationship (collectively **45** and **55**) with the two items ordered objects **50** and **60**, respectively. Order object **30** is an instance of an outstanding order object having a relationship with a single order item, but order object **30** has a 1 to many relationship (**65**) with the item ordered object **70**, since many order items could have been associated. Order object **40** is an instance illustrates a 1 to many relationship (collectively **75** and **85**) with the two items ordered objects **80** and **90**, respectively.

FIG. **2** is a complex data object (CDO) graph drawing, which illustrates a company object and some of its related objects (corporate address object and some of its departments and employees), as well as relationships between the objects. Relationships of all three types: 1 to 1 (1—1), 1 to many (1–M) and many to many (M—M) are shown in this CDO graph. More specifically, FIG. **2** illustrates a CDO graph drawing presenting an instance of a company object **100** having a 1 to 1 relationship (**650**) with its corporate address object **700**, and a 1 to many relationship (collectively **150**, **250**, and **350**) with the three company department objects **200**, **300** and **400**, respectively. Since employees of this company may work for more than one of the company's departments, the company department objects **200**, **300** and **400** in FIG. **2** are three instances (many) of company department objects having relationships (**425**, **450**, **550** and **575**, respectively) with two (many) employee objects (respectively, **500** and **600**). The cross-assignment of employee object **500** to both company department objects **200** and **300**, and of employee object **600** to both company department objects **300** and **400**, illustrate a complex many to many (M—M) relationship of departments to employees for this company.

DISCRIPTION THE INVENTION

The present invention provides a system for creating, maintaining, accessing, navigating and persisting complex data objects as a complex data object graph model. In one embodiment, the system comprises a computer system having a user interface, a working memory storage area and at least one device for permanently storing information, and said system being loaded with at least one portion of a computer software program, wherein said software program comprises at least one user access interface and a set of programming routines designed for creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred embodiment, the present invention provides such a system that can display the CDOG model as a graph or set of tables of complex data objects and provides a creating or editing interface for the CDOG model. Most preferred is such a system providing a point and click graphical user interface.

The present invention provides a method for creating, maintaining, accessing, navigating and persisting complex data objects stores as a CDOG model, comprising the steps of:

a) creating a CDOG representation definition,

b) monitoring and recording any changes to the CDOG on a local or distributed computer system, and

c) persisting any changes to the CDOG on a local or distributed computer system by updating the CDOG or CDOG representation definition to reflect and changes to any copies of the CDOG or CDOG representation definition, and saving a copy of the updated CDOG representation definition to a data source or to another type of information repository.

In a preferred embodiment, the present invention provides such a method providing the additional step of displaying or printing a representation of the CDOG model as a graph or set of tables representing complex data objects and their relationships.

The present invention provides a computer software program having at least one user interface and having at least one feature that provides for at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect of the invention, the present invention provides a software program, or an association with a software program, having a feature for displaying, printing or displaying and printing a representation of the CDOG model as a graph, or as a set of tables representing a CDOG model. In a further preferred embodiment, such a software program has an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the complex data object CDOG model (or has both an interface for editing and an editable input or source, such as a file).

The present invention also provides an application programming interface ("API"), as a preferred embodiment, wherein the API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In one aspect the API can be accessed by an independent computer software program, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, the present invention provides such an API as part of a software tool package that provides a method for displaying or printing a representation of the CDOG model as a graph or set of tables representing the CDOG model (or providing methods for both displaying and printing a representation of the CDOG model). In another preferred aspect, such a software tool package that includes the API provides an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the CDOG model (or provides both an editing interface and an editable input or source, such as a file,).

A preferred embodiment of the present invention provides a software tool comprising the API according (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

In another preferred embodiment, the CDOG API is the CocoNavigator API which provides support for manipulating complex objects in conjunction with the THOUGHT Inc. CocoBase Enterprise Object/Relational (O/R) database

(hereinafter "CocoBase") mapping tool (see U.S. Pat. No. 5,857,197 (incorporated herein in its entirety), for concepts upon which the CocoBase mapping tool is based). An object is considered to be complex when it has established or potential relationships (i.e. links) to other objects. For instance, a Company object instance may be linked to an instance of its corporate address and instances of a Department object, which in turn may be linked to a set of instances of Employee objects. An example of such an CDOG is shown in FIG. **2**, for example. The combination of these objects and their links constitute a graph of objects (CDOG model) that can be manipulated using the CocoNavigator API. Since the CocoNavigator API works in a Java Programming language environment, this CDOG example may also be referred to as a JDOG example.

In another preferred embodiment, the invention provides a software tool comprising the API according to the invention or interfacing therewith, wherein the software is adapted for performing the following steps:

a) reading a source programming object logic model or a database definition file in a format selected from the group consisting of a Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, and an Extensible Markup Language ("XMI") file; and

b) converting the information of (a) into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file.

In a further preferred embodiment, the present invention provides a software tool as described above that is adapted for performing at least one of the following additional steps:

a) displaying a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

b) storing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

c) printing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

d) displaying the target information of (b);

e) storing the target information of (b); and

f) printing the target information of (b).

In a still further preferred embodiment the software tool described above provides target information that is at least one member selected from the group consisting a CocoBase database definition repository file, a CocoBase map, a CocoNavigate CDOG definition, a CocoNavigate object link descriptor, and a CDOG object graph definition data file. Further preferred is such software, wherein the source is a UML/XMI document containing sufficient DTD information to exchange modeling information with a UML modeling tool.

In one embodiment of the software tool according to the invention, the software tool can be set to automatically generate a persistence layer that corresponds to the source UML class diagrams provided by an exported source file from a case tool capable of exporting UML/XMI, such as Rational Rose, Together and ArgoUML.

An easy way to understand the job performed by a CDOG Navigator API, such as the CocoNavigator API, is to imagine it as a monitor that can provide at least one object to intercept any accesses to the data objects and their relationships that constitute the CDOG, as well as any changes to the data source upon which the CDOG depends, in order to provide a means to persist any changes to the CDOG (or CDOG model) or related data in the data source. Any changes to the CDOG (or CDOG model) can then be propagated by the CDOG Navigator API to a persistent data source through a CocoBase Enterprise O/R connection. Similarly, persistent data that updates a data source can be utilized to create a CDOG model or to update a CDOG model.

Each CDOG (or CDOG model) managed by the CDOG Navigator API can be associated by the CDOG Navigator API with a CDOG descriptor (such as a file) that may be utilized to represent all or part of a "navigation model". In this respect, a navigation model may be abstractly described as essentially a namespace in which a set of data objects and a set of link descriptions (i.e., relationship types between data objects) are each defined. In a preferred embodiment, a data source, data field, data field size and data field type are listed for each data object in the set of data objects. In another preferred embodiment, at least one link description for two data objects, or for a single data object having a circular link (where the link description is utilized in conjunction with the CocoNavigator API and CocoBase mapping tool) contains one or more of the following types of information (however other related information may be utilized):

- at least one link type (e.g., 1:1, 1:M, M:1 and M:M) between data objects
- at least one link qualifier (currently supported qualifiers are STD and BEAN)
- at least one associative CocoBase map (optional, for M:M links only)
- the names of the CocoBase maps related to each of the data objects (usually two maps that are not necessarily distinct), and
- information that may be obtained for each of the CocoBase maps related to the relationship between two data objects, including:
  - the name of a relationship link between the two data objects
  - the names of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects
  - the name of a data source for each of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects
  - the names of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)
  - the names of a data source for each of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)
  - the name(s) of any java classes corresponding to fields that would impact upon the relationship between two data objects
  - a setting parameter indicating whether the a loading of a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)
  - a setting parameter indicating whether a deletion of information from a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)
  - a setting parameter indicating whether an insertion of information into a particular CocoBase map that will

impact upon a relationship between two data object should be cascaded to other related map(s), and

a setting parameter indicating whether an update of information in a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s).

Some preferred features provided by the CocoNavigator API and its associated software, when utilized with the CocoBase mapping tool, or with a similar mapping tool, are as follows:

(a) provides multiple navigation models for the same CDOG model

A preferred embodiment of the CocoNavigator API or an associated program module is configured to allow many navigation models to be used with the same set of java classes and CocoBase maps that are associated with a CDOG model. This preferred embodiment API, or an associated computer program module, may also be configured to permit a user to switch dynamically from one such navigation model to another. Thus, while a given application is being executed, it is possible to completely change the relationships between data objects and how CDOGs of a CDOG model should be managed.

(b) circular link definitions

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage circular data object links. In a navigation model, a circular link may be defined as occurring when a data object is directly or indirectly linked to itself as part of a CDOG. Such relationships can be created, accessed, supported and correctly managed via the CocoNavigator API.

(c) bi-directional and oriented links

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage links between data objects as either a bi-directional link or as an oriented link. In this embodiment links between data objects are usually bi-directional links and can be navigated back and forth as an un-oriented navigational model of the CDOG model. With such a bi-directional link feature activated, any object in the navigation model in a given CDOG model can be used as a navigation entry point. In an oriented link navigation model, the link may be set as an oriented link accessible from a single direction. Thus, changes must be made from top down or from bottom up on a relationship tree representation of the navigation model, and some actions may need to originate from a pre-set node of the relationship tree. In this embodiment, such types of links can be created, accessed, supported and correctly managed via the CocoNavigator API.

(d) dynamic link proxy properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, is configured to permit a user to populate any object property (e.g., public field or a getter/setter pair) having its type declared as an object class with special dynamic proxies that can monitor and update the state of a relationship link. Examples of such types of object classes are thought.Cocobase.navapi.LinkObjProxy of CocoBase and the Sun Microsystems java class,java.util.Vector. Bi-directional or single directional object references can be consistently maintained in this manner, or by other similar logical mechanisms.

(e) automatic synchronization of object properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to automatically merge a data object and its properties into a CDOG navigation model representation when the class of object being navigated (being created, accessed, supported or maintained) has a property (e.g., a field or a getter/setter pair) with a name matching the name of a corresponding link as defined in the navigation model.

(f) pure object models

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to include a data object as a relationship link to another data object in a navigation model without the need to declare fields and properties for the data object. Such links should be limited however to cases where no computer code dependency will exist between the java classes that will be associated with such linked data objects of the navigation model. Thus, such a navigation model can be a pure abstraction, and more reusable than just a populated version or single hypothetical instance of the abstraction.

(g) customized link visitors

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can provide an interface which can be configured to permit a visitor object to use this interface to visit (e.g., access or change) a data object that is part of a CDOG navigation model. In conjunction with CocoBase, an example of CocoNavigator API implementing this concept might using a LinkVisitor object (class). Other similar classes may be defined for this purpose. The visitor interface provides a way for a user to customize and adapt the software, and thereby allows a user to extend the functionality of the CocoNavigator API, or the functionality of an associated computer program module, to provide a desired customizable behavior in their CDOG navigation model.

(h) distributed environments

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to operate as a tool to create, access, support and correctly manage a CDOG navigation model in a server environment (e.g., in an EJB container) and to persist any changes to the CDOG navigation model when a navigation model is distributed across a local network or when the navigation model involves a distributed network (e.g., a navigation model distributed across internet connections). In one aspect, on the server side of the network, a CDO or a CDOG model of any complexity sent by clients (e.g., serialized copies) across a local network, or across a distributed network, can be correctly merged into a CDOG model by the CocoNavigator API, or by an associated computer program module. Additionally, the CocoNavigator API, or an associated computer program module, can be configured to send a CDO or CDOG model to a client along with link proxies serialized with parts of the CDO or CDOG model that are being monitored by such link proxies. By sending the CDO or CDOG model copy along with such link proxies and associated parts of the CDOG or CDOG model to a client, dynamic link proxies (described above in (d)) can be used by the client side. Accordingly, a CDOG navigation model can be created, accessed, supported, managed and persisted over a distributed network.

(i) group loading or lazy loading of links

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to monitor some or all of the data objects (and associated relationship definitions) participating in links of a CDOG navigation model. In a more preferred embodiment, the data objects (and associated relationship definitions) participating in links of a CDOG navigation model can be loaded as

they are needed (lazy loading) from a persistent data source or from another type of information repository. This lazy loading feature can permit a very large CDOG navigation model to be loaded in a per-link basis as needed. In a more preferred embodiment, the lazy loading feature can be configured to prevent a link from being loaded twice and can be configured to detect cycles of related loads that have been previously loaded. Such configuration adaptability can be exploited to provide the more efficient use of the resources for a computer system or network or for a distributed computer network.

Some examples of preferred features that can be provided by the CocoNavigator API and its associated software, which are particularly enhanced when utilized with the currently existing CocoBase mapping tool, are as follows:

(aa) CocoBase transactions

A preferred embodiment of the CocoNavigator API or an associated program module is configured to bind a data object or data object relationship of a CDOG navigation model with a CocoBase transaction. Such configurations may be utilized to optimize access to the database server and to enhance performance of the computer system, or systems, involved with the CDOG navigation model.

(bb) non-CBProp objects

The CocoNavigator API or an associated program module can be configured to automatically detect non-CBProp objects (objects with classes that do not implement the CocoBase interface known as the CBProp interface), and automatically create proxies in order to persist such objects as part of a CDOG navigation model.

(cc) virtual foreign key fields

The CocoNavigator API or an associated computer program module can be configured to use proxy classes of CocoBase, such as the CocoProxyM classes, when appropriate. Implementing proxy classes such as the CocoProxyM classes can provide a system for creating, accessing, supporting, properly maintaining and persisting virtual foreign key fields (i.e., foreign key fields do not need to be defined in the object class for Java programming implementations) by the CocoNavigator API, or by an associated computer program module.

(dd) transparent persistence

The CocoNavigator API or an associated computer program module can be configured to use a configurable network component capable of acquiring and persisting CDOGs through network APIs and thereby provide persistence transparently with respect to the applications that are using the CDOGs. In a preferred implementation, there is provided a software tool comprising the CocoNavigator API that is adapted for a local network or for a distributed network environment, wherein the software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGS. Such a concept may be referred to as transparent persistence in the context of this API, because persistence can be obtained without requiring reconfiguring of an existing CDOG model or software application in order to provide such persistence to a computer software application.

From the above description of features of the API, and the features of the CocoNavigator API programming routines described below, an object computer language programmer can produce an API have the functionality of the CocoNavigator API described herein. Essentially the CocoNavigator API has an at least one user access interface, at least one data source access interface and at least three main programming modules consisting of the following programming modules or routines:

I. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:
   a) a computer programming sub-routine or sub-module for obtaining from a member selected from the group consisting of a data source, an information repository, and an input device, sufficient information to construct a CDOG model containing one or more CDOGs,
   b) a computer programming subroutine or sub-module for constructing and loading into the memory registers of the computer system a CDOG or a CDOG model representation definition from a data source or other repository, and
   c) a computer programming sub-routine or sub-module for sending a copy of some portion or all of the CDOG representation definition to a user or to a software program module on a local computer system or on a distributed network;

II. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:
   a) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system that has accessed or changed a portion of a CDOG or CDOG model, which is included in the CDOG, or CDOG model, representation definition of (I), above, and obtaining any changes to the CDOG or CDOG model,
   b) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system who has obtained a copy of any portion of the CDOG, or CDOG model, representation definition, and for obtaining a copy of any changes that the user might have made to any portion of the CDOG, or CDOG model, representation definition, and
   c) a computer programming sub-routine or sub-module for comparing a copy of a CDOG, CDOG model, or a representation definition of either the CDOG or CDOG model, to an original stored version of the CDOG, CDOG model, or an original stored representation definition for the CDOG or CDOG model, and for updating the original to incorporate any changes to a CDOG or a representation definition that are made by the user or by a software program module; and

III. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or software program module of the computer system:
   a) a computer programming sub-routine or sub-module for storing a new CDOG or CDOG model, or storing a definition of either the CDOG or CDOG model, in a data source or in another information repository, and
   b) a computer programming sub-routine or sub-module for persisting (saving to permanent storage) either a changed portion of an updated CDOG, an updated CDOG model, or an updated definition representation for either a CDOG or a CDOG model, to a data source or to another type of information repository.

In a more preferred embodiment, the CDOG API according to the invention, (a preferred embodiment is the CocoN-

avigator API) can be written with an interface that accesses and uses functionalities of the CocoBase mapping tool, associated programming routines, or associated class libraries. Both the object code and the source code of the CDOG API are contemplated as part of this invention.

In another preferred embodiment, the above described CDOG API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In a particularly preferred aspect, the API can be accessed by a computer software program such as the CocoBase mapping tool, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In an even more preferred aspect, the present invention provides such an API as part of a software tool package having a feature or method for displaying or printing a graph or set of tables representing a CDOG or a CDOG model. In yet another preferred aspect, such a software tool package including the API provides an interface for editing a CDOG, a CDOG model, or for editing a representation definition for a CDOG or a CDOG model to provide a modification that implements a change to a CDOG, or to its definition.

EXAMPLE 1

```
thought.CocoBase.navapi
Class Navigator
java.lang.Object
        |
        +--thought.CocoBase.navapi.Navigator
        public class Navigator
        extends java.lang.Object
```

This class provides support for manipulating complex objects and links (relationships) with CocoBase.

Overview:

Navigator provides support for manipulating complex objects with CocoBase. A object is considered to be complex when it has established relationships (i.e. links) to other objects. For instance, a object company may be linked to instances of Departament, which in turn may be linked to a set of Employee objects. The combination of these objects and their links contitute a graph of objects that can be manipulated using the Navigator class.

A navigator is basically a container that keeps track of the links between objects and allows operations to be performed at the graph level. Thus, it is possible to load, insert, delete or update an entire graph of objects provided a CocoBase connection is given. An easy way of understanding the job performed by a Navigator is to imagine it a monitor of object graphs. For each of these graphs, the Navigator keeps has a memory image and a persistent image. When the Navigator is first used, both the memory and persistent images are empty. As links are populated, the memory image grows. A method (e.g., updateAllLinks (. . .)) can be then invoked to propagate changes and synchronize the persistent image with the state of the memory image. Conversely, a persistent graph image can be loaded into memory by invoking loadAllLinks (. . .).

The funcionality available in the Navigator class is similar to that available in the old CocoNavigate class, but it is provided here at an instance level. Thus, it is possible to have several Navigator instances associated to the same or different models. Further, all memory resources can be reclaimed by the garbage collector once the instance is no longer needed.

Features implemented in this version:

multiple navigation models —a navigation model is essencially a namespace in which a set of link descriptions is defined. Each model is identified and defined by the properties file that describes the link information and is associated to set of object graphs. A model can be specified by invoking the method setmodel (. . .). Once a model is defined, description contained in the model_name.properties file or in the CocoBase repository (not available yet) is translated into CBLink link descriptors (see class CBLink for details). If no model is specified, a model called "CocoNavLinks" is automatically created from the description contained in CocoNavLinks.properties. It is possible to work with more than one model (and properties files) at the same time and to switch between those models as well. The basic format of a properties file that describes a model is the following:

```
;list of links defined for MapName1
MapName1__LINKS=linkName1, [,linkName1 [, . . .]]
;the list of field labels used as navigation key in
MapName1
MapName1__linkName1__KEY=keyName1 [,keyName2 [, . . . ]]
;the name of the map that maps the target object
MapName1__linkName1__TARGET__MAP=mapName
;the list of field labels used as navigation key in the
target map
MapName1__linkName1__TARGET__KEY=keyName1
[, keyName2 [, . . . ]]
;the full class name of the java class corresponding to
the target map
MapName1__linkName1__TARGET__CLASS=pkgname.class name
;the type of the link (one-to-many, one-to-one, many-to-
one, many-to-many)
MapName1__linkName1__TARGET__TYPE=1__M(default)
|1__1|M__1|M__M
;the qualifier of the link (used for code generation
purposes)
MapName1__linkName1__TARGET__QUALIFIER=STD__LINK(default)
|BEAN__LINK
;will the target object be automatically loaded (use false
for "lazy" load)
MapName1__linkName1__AUTONAV=true)default) |false
;will this link cascade deletion to the target object?
MapName1__linkName1__TARGET__DELETE=true(default) |false
;will this link cascase updates to the target object?
MapName1__linkName1__TARGET__UPDATE=true|false(default)
;will this link cascade insertion to the target object?
MapName1__linkName1__TARGET__INSERT=true(default) |false
;specify this link definition as inverse of another
(optional when type is 1__1, 1__M, M__1)
MapName1__linkName1__INVERSE__OF=mapName__linkName
;the name of the associative map for many-to-many links
(requires __INVERSE__OF entry)
MapName1__linkName1__M__TO__M__MAP=mapName
```

EXAMPLE

```
. . .
Department__LINKS=employees
Department__employees__KEY=ID
Department__employees__TARGET__KEY=DEPARTMENT
Department__employees__TARGET__MAP=Employee
Department__employees__TARGET__CLASS=com.foo.Employee
Department__employees__TARGET__TYPE=1__M
Department__employees__TARGET__QUALIFIER=STD__LINK
Department__employees__AUTONAV=true
Department__employees__TARGET__DELETE=true
Department__employees__TARGET__INSERT=true
Department__employees__TARGET__UPDATE=true
. . .
Employee__LINKS=dependents
. . .
```

circular references are supported —in the object model, a circular reference or cycle occurs when it is possible for object to directly or indirectly refer to itself. Such models are supported and correctly managed by the INVERSE_OF keyword that can be included in the properties file. Note that "inverse" links between objects are inherently bi-directional and can be navigated back and forth, as in a unoriented graph (as long as navigation is enabled in both directions). This is a very important feature since it allows any object in a given graph to be used as an navigation entry point.

```
. . .
Employee_LINKS=department
. . .
Employee_department_INVERSE_OF=Department_employees
. . .
Employee_LINKS=employees
. . .
Department_employees_INVERSE_OF=Employee_department
. . .
```

support for dynamic fields —upon loading a graph, any object property (i.e. public field or getter/setter pair) that has the type declared as a java. util. Vector or as a thought.Cocobase.navapi.LinkobjProxy will be populated with special dynamic proxies that monitor and update the state of a link, provided the property name matches the link name of the corresponding link. This allows bi-directional references to be automatically maintained in a consistent way. For example:

```
class Employee implements CBProp {
  LinkObjProxy department;
. . .
}
class Department implements CBProp {
  Vector employees;
   . . .
}
Navigator n = new Navigator (myBase);
Department hardware = myBase.select (new
Department (100, "HARDWARE", aCompany));
// load a graph of objects from a given department
n.loadAllLinks (hardware);
// add this point hardware.employees is populated with a
special dynamic Vector
Employee john = new Employee (10010, "John", "123 Main St");
// at this point john.department is null and
hardware.employees.contains(john)==false
hardware.employees.addElement(john);
// at this point john.department is a special dynamic
proxy containing hardware
// and hardware.employees.contains(john)==true
john.department.set (null);
// at this point john.department.get( )==null and
hardware.employees.contains(john)==false
n.addLink(john, "department", hardware);
// at this point john.department.get( )==hardware
// and hardware.employees.contains(john)==true
. . .
```

In the example above, if an Employee object is added to the vector employees of a Department object, it will have that department automatically contained in the proxy field department. Conversely, if that Employee object sets the object contained by the proxy field department to null, the department will be automatically removed from the Vector employees of the corresponding department. Also, any calls to addLink (. . .) and removeLink(. . .) will have the result reflected in the corresponding fields.

automatic synchronization of object properties —if the class of the object being used as an entry point for navigation has a property (field or getter/setter pair) with a name matching a the name of the corresponding link as defined in the model, the contents of such property will be automatically merged with the object graph. So if, for example, in the case where references between objects are created "manually" (i.e. links are not automatically populated by the loadAllLinks method) and updateAllLinks (. . .) is invoked, before the update actually occurs, the references between the objects will be identified and the corresponding links will be created and merged into the object graph.

In the code bellow, the references between anEmp and aChild through the field dependends will be automatically identified (since the referring field has the same name as the link name defined for the class Employee in the navigation model) and will be added to the graph. As a result, a call to updateAllLinks (. . . ,anEmp, . . .) will provoke the insertion of both anEmp and aChild to the memory and persistent images of that graph (note that the navigation model mast have cascade insertion enabled in the direction Employee->dependents for that to happen):

```
public class Employee {
      String name;
      Vector dependents;
      . . .
}
public class Dependent {
      String name;
      Employee parent;
      . . .
}
. . .
Employee anEmp = new Employee ("John Foo");
Dependent aChild = new Dependent ("Lori Foo");
anEmp.dependents = new java.util.Vector( );
anEmp.dependents.addElement (aChild);
// at this point aChild.parent==null and
anEmp.dependents.contains (aChild)==true
. . .
Navigator n = new Navigator (myBase);
n.updateAllLinks (anEmp, "Department", true);
// at this point the object graph contains a link between
anEmp and aChild
// and aChild.parent==anEmp
. . .
```

support for pure object models —objects do not necessarily have to declare fields/properties in order to establish links with other objects. Instead, the methods addLink(. . .),getAllLinkedTo(. . .)and removeLink(. . .) can be used for that purpose. Thus, the object model can be pure and more reusable, provided no code dependency will exist between the classes. The concept of virtual foreign key fields (i.e. foreign key fields do not need to be defined in the object class) and non-CBProp object classes (i.e. classes that do not implement CBProp) are also supported for that same purpose.

customized link visitors —objects in the object graph can be visited by visitor objects (i.e. objects that implement the LinkVisitor interface), thus allowing the functionality to be extended with customized behavior. For example, the code bellow creates a visitor object that prints contents of the visited link so that the whole object graph can be printed by invoking the method visitAllLinks(. . .).

```
. . .
n.visitAllLinks (hardware,
  new LinkVisitor( ) {
  public void
```

-continued

```
visitLink (CocoDriverInterface cdi,
                                        CBLink
cbLink,
                                        Object obj,
                                        String
mapName,
                                        String
linkName,
                                        Object
target,
                                        int
distance,
                                        boolean
hasBeenVisited) {
   for (int i=0; i < distance+1;
i++) System.out.pring ("\t"); // we use tabs to indicate the
level in the tree
   System.out.println ("["+
linkName +"]" + target + (hasBeenVisited ? "++" : ""));
   }
  };
  false);
. . .
```

Complete usage example:

Assume the following object model:

```
class Employee implements CBProp {
    long id;
    String name;
    Employee lmanager;
    Vector lsubordinates;
    LinkObjProxy ldepartment;
        . . .
}
class Department implements CBProp {
    long id;
    String name;
    Vector lemployees;
    LinkObjProxy lcompany;
        . . .
}
class Company implements CBProp {
    long id;
    String name;
    Vector ldepartments;
        . . .
}
```

Suppose the following database description (assume each table has a corresponding map definition) for the object model given above:

```
CREATE TABLE COMPANY (
    ID NUMERIC (15),
    NAME VARCHAR (15) )
CREATE TABLE DEPARTMENT (
    ID NUMERIC (15),
    NAME VARCHAR (15),
    COMPANY NUMERIC (15 ) )
CREATE TABLE EMPLOYEE (
    ID NUMERIC (15),
    NAME VARCHAR (15),
    DEPARTMENT NUMERIC (15),
    MANAGER NUMERIC (15) )
```

The CocoNavLinks.properties file is something like:

```
#This is the navigation properties file
Company__LINKS=ldepartments
Company__ldepartments__KEY=ID
Company__ldepartments__TARGET__KEY=COMPANY
Company__ldepartments__TARGET__MAP=Department
Company__ldepartments__TARGET__CLASS=com.foo.Department
Company__ldepartments__TARGET__TYPE=1__M
Company__ldepartments__TARGET__QUALIFIER=STD__LINK
Company__ldepartments__AUTONAV=true
Company__ldepartments__TARGET__DELETE=true
Company__ldepartments__TARGET__UPDATE=false
Company__ldepartments__TARGET__INSERT=true
Company__ldepartments__INVERSE__OF=Department__lcompany
Department__LINKS=lcompany, lemployees
Department__lcompany__KEY=COMPANY
Department__lcompany__TARGET__KEY=ID
Department__lcompany__TARGET__MAP=Company
Department__lcompany__TARGET__CLASS=com.foo.Company
Department__lcompany__TARGET__TYPE=M__1
Department__lcompany__TARGET__QUALIFIER=STD__LINK
Department__lcompany__AUTONAV=true
Department__lcompany__TARGET__DELETE=false
Department__lcompany__TARGET__UPDATE=false
Department__lcompany__TARGET__INSERT=false
Department__lcompany__INVERSE__OF=Company__ldepartments
Department__lemployees__KEY=ID
Department__lemployees__TARGET__KEY=DEPARTMENT
Department__lemployees__TARGET__MAP=Employee
Department__lemployees__TARGET__CLASS=com.foo.Employee
Department__lemployees__TARGET__TYPE=1__M
Department__lemployees__TARGET__QUALIFIER=STD__LINK
Department__lemployees__AUTONAV=true
Department__lemployees__TARGET__DELETE=true
Department__lemployees__TARGET__INSERT=true
Department__lemployees__TARGET__UPDATE=true
Department__lemployees__INVERSE__OF=Employee__ldepartment
Employee__LINKS=ldepartment, lmanager, lsubordinates
Employee__ldepartment__KEY=DEPARTMENT
Employee__ldepartment__TARGET__KEY=ID
Employee__ldepartment__TARGET__MAP=Department
Employee__ldepartment__TARGET__CLASS=com.foo.Department
Employee__ldepartment__TARGET__TYPE=M__1
Employee__ldepartment__TARGET__QUALIFIER=STD__LINK
Employee__ldepartment__AUTONAV=true
Employee__ldepartment__TARGET__DELETE=false
Employee__ldepartment__TARGET__UPDATE=false
Employee__ldepartment__TARGET__INSERT=false
Employee__ldepartment__INVERSE__OF=Department__lemployees
Employee__lmanager__KEY=MANAGER
Employee__lmanager__TARGET__KEY=ID
Employee__lmanager__TARGET__MAP=Employee
Employee__lmanager__TARGET__CLASS=com.foo.Employee
Employee__lmanager__TARGET__TYPE=M__1
Employee__lmanager__TARGET__QUALIFIFER=STD__LINK
Employee__lmanager__AUTONAV=true
Employee__lmanager__TARGET__DELETE=false
Employee__lmanager__TARGET__UPDATE=false
Employee__lmanager__TARGET__INSERT=false
Employee__lmanager__INVERSE__OF=Employee__lsubordinates
Employee__lsubordinates__KEY=ID
Employee__lsubordinates__TARGET__KEY=MANAGER
Employee__lsubordinates__TARGET__MAP=Employee
Employee__lsubordinates__TARGET__CLASS=com.foo.Employee
Employee__lsubordinates__TARGET__TYPE=1__M
Employee__lsubordinates__TARGET__QUALIFIFER=STD__LINK
Employee__lsubordinates__AUTONAV=true
Employee__lsubordinates__TARGET__DELETE=false
Employee__lsubordinates__TARGET__INSERT=true
Employee__lsubordinates__TARGET__UPDATE=false
Employee__lsubordinates__INVERSE__OF=Employee__lmanager
```

Let us now use the Navigator class accordingly:

```
. . .
Company company = myBase.select (new
Company (1, "FOO"), "Company");
n = new Navigator (myBase)
// load a graph of objects from a given department
n.loadAllLinks (company, "Company");
int hdwidx = company.ldepartments.indexOf (new
Department (100, "hardware"));
Department hardware =
company.ldepartments.elementAt (hdwIdx);
// let us make some changes to the graph using the proxy
fields
Employee john = new Employee (10010, "John Foo", "123 Main
St");
hardware.lemployees.addElement (john);
// at this point, john.ldepartment.get ( ) == hardward and
// hardware.lemployees.contains (john) == true
// let us remove some of the links using the proxy fields
hardward.lemployees.removeElementAt (0);
company.ldepartments.removeElement (new
Department (200, "ACCOUNT"));
// let us update all changes
n.updateAllLinks (company, "Company");
// let us now make some more changes to the graph using
the navigator
mary = new Employee (10010, "John Foo", "123 Main
St", hardware);
// the line below will create a link between "mary" and
"hardware" and automatically
// add "mary" to the "hardward.lemployees" Vector and
// assign "hardware" to the "mary.ldeparment" field
n.addLink (mary, "Employee", "ldeparment", hardware);
// the line below has no effect since "mary" and
"hardward" are already linked
n.addLink (hardware, "Department", "lemployees", mary);
// let us set john as mary's supervisor
n.addLink (john, "Employee", "lsubordinates", mary);
// we can also retrieve all employees subordinated to john
using the navigator
Object [ ] johnsSubordinates =
n.getAllLinkedTo (john, "Employee", "lsubordinates");
// and remove one of his subordinates
Employee fstJohnSubordinate = (Employee)
john.subordinates.firstElement( );
n.removeLink (fstJohnSubordinate, "Employee", "lmanager", john)
;
// which has the exact same effect as
n.removeLink (john, "Employee", "lsubordinates", fstJohnSubordinate);
// or as removing from the Vector directly
john.subordinates.removeElement (fstJohnSubordinate);
// let us now update all changes since the last call to
updateAllLinks
n.updateAllLinks (company);
// at this point both the memory and persistent graph
images are synchronized
. . .
```

See Also:

thought.CocoBase.navapi, CBLink

Constructor Summary

Navigator ( )

Constructs a Navigator instance using the default model

Navigator (CocoDriverInterface cdi)

Constructs a Navigator instance using the default model and sets its CocoBase connection to a persistent data source.

Navigator (CocoDriverInterface cdi, java.lang.String model)

Constructs a Navigator instance using the given model and sets its CocoBase connection to a persistent data source.

Navigator (java.lang.String model)

Constructs a Navigator instance using the given model.

TABLE 1

Method Summary	
void	addLink (java.lang.Object object, java.lang.String linkName, java.lang.Object target) Invokes addLink (Object, String, String, Object) sending object.getClass ( ).getName ( ) as the map name.
void	addLink (java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, java.lang.Object target) Establishes a link between two objects in the object graph of this Navigator
void	clear ( ) Clears the memory and persistent graph images held by this Navigator and resets all information about loaded links.
int	deleteAllLinks (java.lang.Object object) Invokes deleteAllLinks (Object, String, boolean) sending object.getClass ( ).getName ( ) as the map name and true as the deleteRoot flag.
int	deleteAllLinks (java.lang.Object object, java.lang.String mapName) Invokes deleteAllLinks (Object, String, boolean) sending true as the deleteRoot flag.
int	deleteAllLinks (java.lang.Object obj, java.lang.String mapName, boolean deleteRoot) Deletes all objects linked to a given object, recusively, from the persistent data source.
java.lang.Object	findObject (java.lang.Object object, java.lang.String mapName) Finds the object instance in the memory graph image of this Navigator that is equal to the given object.
java.lang.Object [ ]	getAllLinkedTo (java.lang.Object object, java.lang.String linkName) Invokes getAllLinkedTo (Object, String, String) sending object.getClass ( ).getName ( ) as the map name.
java.lang.Object [ ]	getAllLinkedTo (java.lang.Object obj, java.lang.String mapName, java.lang.String linkName) Returns an array containing the objects linked to a given object through a given link.
CocoDriverInterface	getConnection ( ) Returns the CocoBase connection to a persistent data source associated to this Navigator
static int	getDebugLevel ( ) Returns the current debug level.
static Navigator	getInstance (java.lang.String instanceId) Returns a Navigator object registered in the internal static pool.
java.lang.String	getModel ( ) Returns the current model associated to this Navigator.
static java.util.Enumeration	getRegisteredIds ( ) Returns an Enumeration object containing with all the ids of instances registered in the internal static pool.
Transaction	getTransaction ( ) Returns the CocoBase transaction this Navigator should use to bind objects when updating the persistent data source.
int	insertAllLinks (java.lang.Object object) Invokes insertAllLinks (Object, String, boolean) sending object.getClass ( ).getName ( ) as the map name and true as the insertRoot flag.
int	insertAllLinks (java.lang.Object object, java.lang.String mapName) Invokes insertAllLinks (Object, String, boolean) sending true as the insertRoot flag.

TABLE 1-continued

Method Summary	
int	insertAllLinks (java.lang.Object obj, java.lang.String mapName, boolean insertRoot) Inserts all objects linked to a given object recusively, in the persistent data source.
java.lang.Object	loadAllLinks (java.lang.Object object) Invokes loadAllLinks (Object, String, boolean) sending object.getClass ( ).getName ( ) as the map name and true as the bindObjects flag
java.lang.Object	loadAllLinks (java.lang.Object object, java.lang.String mapName) Invokes loadAllLinks (Object, String, boolean) sending true as the bindObjects flag.
java.lang.Object	loadAllLinks (java.lang.Object obj, java.lang.String mapName, boolean bindObjects) Loads all objects linked to a given object, recusively, from the persistent data source.
java.lang.Object	loadLink (java.lang.Object object, java.lang.String mapName, java.lang.String linkName) Invokes loadLink (Object, String, String, boolean, boolean) sending object.getClass ( ).getName ( ) as the map name, true as the bindObjects flag and true as the forceLoad flag.
java.lang.Object	loadLink (java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, boolean bindObjects, boolean forceLoad) Loads all objects linked to a given object through the specified link from the persistent data source.
java.lang.Object	loadLinks (java.lang.Object object) Invokes loadLinks (Object, String, boolean) sending object.getClass ( ).getName ( ) as the map name and true as the bindObjects flag.
java.lang.Object	loadLinks (java.lang.Object object, java.lang.String mapName) Invokes loadLinks (Object, String, boolean) sending true as the bindObjects flag.
java.lang.Object	loadLinks (java.lang.Object obj, java.lang.String mapName, boolean bindObjects) Loads all objects linked to a given object from the persistent data source.
static void	registerInstance (java.lang.String instanceID, Navigator nav) Adds a Navigator object to the internal static pool.
void	removeLink (java.lang.Object object, java.lang.String linkName, java.lang.Object target) Invokes removeLink (Object, String, String, Object) sending object.getClass ( ).getName ( ) as the map name.
void	removeLink (java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, java.lang.Object target) Removes a link between two objects.
void	removeLinks (java.lang.Object obj) Invokes removeLinks (Object, String) sending object.getClass ( ).getName ( ) as the map name.
void	removeLinks (java.lang.Object obj, java.lang.String mapName) Removes an object from the object graph by removing all existing links to that object.
void	setConnection (CocoDriverInterface cdi) Associates the CocoBase connection to a persistent data source with this Navigator.
static void	setDebugLevel (int level) Sets the current debug level.
void	setModel (java.lang.String modelName) Sets the model to be used by this Navigator.
void	setProperties (java.util.Properties properties) Sets the properties that control the behavior of this Navigator.

TABLE 1-continued

Method Summary	
void	setTransaction (Transaction transaction) Sets the transaction to which objects should be bound by this Navigator when updating the persistent data source.
static void	unregisterInstance (java.lang.String instanceId) Removes a Navigator object from the internal static pool.
int	updateAllLinks (java.lang.Object object) Invokes updateAllLinks (Object, String, boolean) sending object.getClass ( ).getName ( ) as the map name and true as the updateRootflag.
int	updateAllLinks (java.lang.Object object, java.lang.String mapName) /**Invokes updateAllLinks (Object, String, boolean) sending true as the updateRoot flag.
int	updateAllLinks (java.lang.Object obj, java.lang.String mapName, boolean updateRoot) Updates the persistent data source with all changes made to memory graph image that are not yet propagated to the persistent graph image.
int	visitAllLinks (java.lang.Object object, LinkVisitor visitor, boolean sync) Invokes visitAllLinks (Object, String, LinkVisitor, boolean) sending object.getClass ( ).getName ( ) as the map name.
int	visitAllLinks (java.lang.Object obj, java.lang.String mapName, LinkVisitor visitor, boolean sync) Visits a given object and invokes visitor.visitLink (. . .) for each of the target objects linked to it, recusively.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Navigator Constructor Examples

public Navigator( )

Constructs a Navigator instance using the default model

public Navigator (CocoDriverInterface cdi)

Constructs a Navigator instance using the default model and sets its CocoBase connection to a persistent data source.

Parameters:

cdi —the CocoBase connection to a persistent data source

public Navigator(CocoDriverInterface cdi, java.lang.String model)

Constructs a Navigator instance using the given model and sets its CocoBase connection to a persistent data source.

Parameters:

cdi —the CocoBase connection to a persistent data source

model —the CocoBase connection to a persistent data source

public Navigator(java.lang.String model)

Constructs a Navigator instance using the given model.

Parameters:

model —the CocoBase connection to a persistent data source

Navigation Method Detail

addLink

public void addLink	(java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, java.lang.Object target)

Establishes a link between two objects in the object graph of this Navigator. Does not have any effect if the link already exists or if a corresponding link is not defined in the current model. In order to make the link persistent, either the method updateAllLinks . . .) or the method insertAllLinks(. . .) must be invoked.

If the class of the object being linked has a property (getter/setter pair or public field) with a name that matches the given link name, that property will be set with the object being linked. Also, any "special" Vector or proxy fields that populate such object property will be notified to add the new linked object.

Parameters:

obj—the source object

mapName —the map name corresponding to the class of obj

linkName —the name of the link that connects obj to the target

target —the target object

See Also:

updateAllLinks(Object,String,boolean), insertAllLinks (Object, String)

deleteAllLinks

public int deleteAllLinks	(java.lang.Object object, java.lang.String mapName)

Invokes deleteAllLinks(Object,String,boolean) sending true as the deleteRoot flag.

Parameters:

object —the object used as entry-point for navigating in the object graph

mapName —the map name corresponding to the class of object

Returns:

the number of objects deleted

See Also:

deleteAllLinks(Object,String,boolean)

getAllLinkedTo

public java.lang.Object[ ] getAllLinkedTo	(java.lang.Object obj, java.lang.String mapName) java.lang.String linkName)

Returns an array containing the objects linked to a given object through a given link. An empty array is returned if no linked object is found or if a corresponding link is not defined in the current model.

Parameters:

obj —Object the source object

mapName —String the map name corresponding to the class of obj

linkname —String the link name that identifies the link

Returns:

Object[] the array of linked objects

insertAllLinks

public int insertAllLinks	(java.lang.Object object, java.lang.String mapName)

Invokes insertAllLinks (Object, String,boolean) sending true as the insertRoot flag.

Parameters:

object —the object used as entry-point for navigating in the object graph

mapName —the map name corresponding to the class of object

Returns:

the number of objects inserted

See Also:

insertAllLinks(Object,String,boolean)

loadAllLinks

public java.lang.Object loadAllLinks	(java.lang.Object object, java.lang.String mapName)

Invokes loadAllLinks(Object,String,boolean) sending true as the bindObjects flag.

Parameters:

object —the object used as entry-point for navigating in the object graph

mapName —the map name corresponding to the class of object

Returns:

the modified instance of the object passed in

See Also:

loadAllLinks(Object,String,boolean)

removeLink

public void removeLink	(java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, java.lang.Object target)

Removes a link between two objects. Does not have any effect if the link does not exist or if a corresponding link is not defined in the current model.

If the class of the object being unlinked has a property (getter/setter pair or public field) with a name that matches the given link name, any "special" proxy fields that populate such object property will be notified to remove the unlinked object.

Parameters:

obj —the source object

mapName —the map name corresponding to the class of obj

linkName —the name of the link that connects obj to the target

target —the target object

See Also:
updateAllLinks(Object,String,boolean)
updateAllLinks

public int updateAllLinks	(java.lang.Object object, java.lang.String mapName)

/**InvokesupdateAllLinks(Object,String,boolean) sending true as the updateRoot flag.
Parameters:
object —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of object
Returns:
the total number of objects updated, deleted or inserted
See Also:
updateAllLinks(Object,String,boolean)
visitAllLinks

public int visitAllLinks	(java.lang.Object obj, java.lang.String mapName, LinkVisitor visitor, boolean sync)

Visits a given object and invokes visitor. visitLink(. . .) for each of the target objects linked to it, recusively. This method is provided as a generic mechanism to implement customized behavior that can be aplied to all objects of an object graph that are reachable from a given navigation entry point. If this method is used to propagate changes to a persistent data source, the flag sync should be set to true.
Parameters:
obj —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of obj
visitor —LinkVisitor the visitor object
sync —boolean indicates if the memory image graph should be synchronized the persistent image in the end of the process
Returns:
int the number of objects visited
addLink

public void addLink	(java.lang.Object object, java.lang.String linkName, java.lang.Object target)

Invokes addLink(Object,String,String,Object) sending object.getclass( ) .getname( ) as the map name.
Parameters:
object —the source object
linkName —the name of the link that connects object to the target
target —the target object
See Also:
addLink(Object,String,String,Object)
deleteAllLinks
public int deleteAllLinks(java.lang.Object object)
Invokes deleteAllLinks (Object, String, boolean) sending object.getClass( ) .getName( ) as the map name and true as the deleteroot flag.
Parameters:
object —the object used as entry-point for navigating in the object graph
Returns:
the number of objects deleted
See Also:
deleteAllLinks(Object,String,boolean)
deleteAllLinks

public int deleteAllLinks	(java.lang.Object obj, java.lang.String mapName, boolean deleteRoot)

Deletes all objects linked to a given object, recusively, from the persistent data source. The object given as parameter will also be deleted if the deleteRoot flag is true. Note that only the objects to which deletion should cascade (according to the current model) will be deleted. In case a CocoBase Transaction is defined, the objects will be bound to be deleted instead. The deleted object will also be removed from object graph and both persistent and memory graph images will be synchronized to the persistent data source with respect to the deleted objects. Note that objects to be inserted will not be added to the persistent image so that a subsequent call to insertAllLinks(. . .) or updateAllLinks(. . .) will insert such objects.
If the class of the object given as parameter has a property (getter/setter pair or public field) with a name that matches a given link name, the referred objects reachable by that link will recursively "merge" into the object graph before the deletion phase takes place. Also, any "special" Vector or proxy fields that populate such object property will be notified to remove the deleted links.
Parameters:
obj —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of obj
deleteRoot —indicates if obj must be deleted
Returns:
the number of objects deleted
See Also:
insertAllLinks(Object,String,boolean), updateAllLinks (Object,String,boolean)
getAllLinkedTo

public java.lang.Object [ ] getAllLinkedTo	(java.lang.Object object, java.lang.String linkName)

Invokes getAllLinkedTo(Object, String, String) sending object.getclass) getName( ) as the map name.
Parameters:
object —the source object
linkName —the name of the link that connects object to the target linked objects
Returns:
the array containing the linked objects
See Also:
getAllLinkedTo(Object, String, String)
getConnection
public CocoDriverInterface getconnection ( )
Returns the CocoBase connection to a persistent data source associated to this Navigator.

Returns:
CocoDriverInterface the CocoBase connection
getDebugLevel
public static int getDebugLevel( )
Returns the current debug level. 0 means debug is not enabled.
Returns:
the debug level
getModel
public java.lang.String getModel( )
Returns the current model associated to this Navigator.
Returns:
the String representing the current model
insertAllLinks
public int insertAllLinks(java.lang.Object object)
Invokes insertAllLinks (Object, String, boolean) sending object. getClass ( ) .getName ( ) as the map name and true as the insertRoot flag.
Parameters:
object —the object used as entry-point for navigating in the object graph
Returns:
the number of objects inserted
See Also:
insertAllLinks(Object,String,boolean)
insertAllLinks

public int insertAllLinks	(java.lang.Object obj, java.lang.String mapName, boolean insertRoot)

Inserts all objects linked to a given object, recusively, in the persistent data source. The object given as parameter will also be inserted if the insertRoot flag is true. Note that only the objects to which insertion should cascade (according to the current model) and that are not yet in the persistent graph image will be inserted. In case a CocoBase Transaction is defined, the objects will be bound to be inserted instead. After a successful invocation of this method, the persistent and memory graph images will be synchronized to the persistent data source with respect to the deleted objects. Note that objects to be deleted will not be included in the persistent image so that a subsequent call to deleteAllLinks(. . .) or updateAllLinks (. . .) will delete such objects.

If the class of the object given as parameter has a property (getter/setter pair or public field) with a name that matches a given link name, the referred objects reachable by that link will recursively "merge" into the object graph before the insertion phase takes place.

Parameters:
cdi —the CocoBase connection
obj —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of obj
insertRoot —indicates if obj must be inserted
Returns:
the number of objects inserted
See Also:
deleteAllLinks(Object,String,boolean), updateAllLinks (Object,String,boolean)
loadAllLinks
public java.lang.Object loadAllLinks(java.lang.Object object)
Invokes loadAllLinks (Object,String,boolean) sending object.getclass( ) .getName( ) as the map name and true as the bindobjects flag.
Parameters:
object —the object used as entry-point for navigating in the object graph
Returns:
the modified instance of the object passed in
See Also:
loadAllLinks(Object,String,boolean)
loadAllLinks

public java.lang.Object loadAllLinks	(java.lang.Object obj, java.lang.String mapName, boolean bindObjects)

Loads all objects linked to a given object, recusively, from the persistent data source. Note that only the objects to which load should cascade (according to the current model) will be loaded. If a CocoBase transaction is defined and the flag bindObjects is true, the loaded objects will be bound to that transaction. All objects loaded will be add to both the memory and persistent graph images. The links followed will be marked as visited for the object being loaded and will not be followed again for the same object. Thus, any subsequent call to loadAllLinks(. . .),loadLinks(. . .) or loadLink(. . .) for the object being loaded will have no effect on the visited links.

If the class of the object being loaded has a property (getter/setter pair or public field) with a name that matches a given link name, that object property will be set with the object loaded through that link. Also, if such property is declared as a java. util. Vector or a LinkObj Proxy, a special "dynamic" proxy will be used to populate the property, so that it can not only be notified about changes in the link but also propagate such changes to other proxies monitoring that same link.

Parameters:
obj —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of obj
bindObjects —indicates if the loaded objects will be bound to the current CocoBase transaction
Returns:
the modified instance of the object passed in
See Also:
loadLinks(Object,String,boolean), loadLink(Object,String, String,boolean,boolean),LinkObjProxy
loadLink

public java.lang.Object loadLink	(java.lang.Object object, java.lang.String linkName)

Invokes loadLink(Object,String,String,boolean,boolean) sending object.getclass ( ) .getName( ) as the map name, true as the bindObjects flag and true as the forceLoad flag.
Parameters:
object —the object of which the given link will be loaded
linkName —the name of the link to be followed
Returns:
the modified instance of the object passed in

See Also:
loadAllLinks(Object,String,boolean), loadLinks(Object, String,boolean)
loadLink

public java.lang.Object loadLink	(java.lang.Object object, java.lang.String mapName, java.lang.String linkName)

Invokes loadLink (Object, String, String,boolean,boolean) sending true as the bindobjects flag and true as the forceLoad flag.
Parameters:
object —the object of which the given link will be loaded
mapName —the map name corresponding to the class of object
linkName —the name of the link to be followed
Returns:
the modified instance of the object passed in
See Also:
loadAllLinks(Object,String,boolean), loadLinks(Object, String,boolean)
loadLink

public java.lang.Object loadLink	(java.lang.Object obj, java.lang.String mapName, java.lang.String linkName, boolean bindObjects, boolean forceLoad)

Loads all objects linked to a given object through the specified link from the persistent data source. If forceLoad is false, the objects will be loaded only if load should cascade to those objects (according to the current model). However, forceLoad is true, the link will be followed regardless of the direction specified in the model. If a CocoBase transaction is defined and the flag bindObjects is true, the loaded objects will be bound to that transaction. All objects loaded will be add to both the memory and persistent graph images. The given link will be marked as visited for the object being loaded and will not be followed again for the same object. Thus, any subsequent call to loadAllLinks (. . .), loadLinks(. . .) or loadLink(. . .) for the object being loaded will have no effect on the given link.
If the class of the object given as parameter has a property (getter/setter pair or public field) with a name that matches a given link name, that object property will be set with the object loaded through that link. Also, if such property is declared as a java. util. Vector or a LinkObjProxy, a special "dynamic" proxy will be used to populate the property, so that it can not only be notified about changes in the link but also propagate such changes to other proxies monitoring that same link.
Parameters:
obj —the object of which the given link will be loaded
mapName —the map name corresponding to the class of obj
linkName —the name of the link to be followed
bindobjects —indicates if the loaded objects will be bound to the current CocoBase transaction
forceLoad —indicate if the link should be followed regardless of the direction specified in the model
Returns:
the modified instance of the object passed in
See Also:
loadAllLinks(Object,String,boolean), loadLinks(Object, String,boolean),LinkObjProxy
loadLinks

public java.lang.Object loadLinks(java.lang.Object object)

Invokes loadLinks(Object,String,boolean) sending object-.getclass ( ) .getName( ) as the map name and true as the bindObjects flag.
Parameters:
object —the object of which links will be loaded
Returns:
the modified instance of the object passed in
See Also:
loadLinks(Object,String,boolean)
loadLinks

public java.lang.Object loadLinks	(java.lang.Object object, java.lang.String mapName)

Invokes loadLinks (Object, String, boolean) sending true as the bindObjects flag.
Parameters:
object —the object of which links will be loaded
mapName —the map name corresponding to the class of object
Returns:
the modified instance of the object passed in
See Also:
loadLinks(Object,String,boolean)
loadLinks

public java.lang.Object loadLinks	(java.lang.Object obj, java.lang.String mapName, boolean bindObjects)

Loads all objects linked to a given object from the persistent data source. Note that only the objects to which load should cascade (according to the current model) will be loaded and that these objects will NOT be processed recursively. If a CocoBase transaction is defined and the flag bindObjects is true, the loaded objects will be bound to that transaction. All objects loaded will be add to both the memory and persistent graph images. The links followed will be marked as visited for the object being loaded and will not be followed again for the same object. Thus, any subsequent call to loadAllLinks (. . .), loadLinks(. . .) or loadLink(. . .) for the object being loaded will have no effect on the visited links.
If the class of the object given as parameter has a property (getter/setter pair or public field) with a name that matches a given link name, that object property will be set with the object loaded through that link. Also, if such property is declared as a java. util. Vector or a LinkObj Proxy, a special "dynamic" proxy will be used to populate the property, so that it can not only be notified about changes in the link but also propagate such changes to other proxies monitoring that same link.
Parameters:
obj —the object of which links will be loaded
mapName —the map name corresponding to the class of obj
bindobjects —indicates if the loaded objects will be bound to the current CocoBase transaction

Returns:
the modified instance of the object passed in
See Also:
loadAllLinks(Object,String,boolean), loadLink(Object, String,String,boolean,boolean),LinkObjProxy
removeLink

public void removeLink	(java.lang.Object object, java.lang.String linkName, java.lang.Object target)

Invokes removeLink(Object,String,String,Object) sending object.getClass( ) getName( ) as the map name.
Parameters:
object —the source object
linkName —the name of the link that connects object to the target
target —the target object
See Also:
removeLink (Object, String, String,Object)
removeLinks

public void removeLinks	(java.lang.Object obj, java.lang.String mapName)

Removes an object from the object graph by removing all existing links to that object. Linked objects are not followed, that is, the removal of links is NOT a recusive procedure.
If the class of the object being unlinked has a property (getter/setter pair or public field) with a name that matches the name of the link being removed, any "special" Vector fields that populate such object property will be notified to remove the unlinked objects.
Parameters:
obj —Object the source object
mapName —String the map name corresponding to the class of obj
setConnection
public void setConnection(CocoDriverInterface cdi)
Associates the CocoBase connection to a persistent data source with this Navigator.
Parameters:
cdi —thought.CocoBase.CocoDriverInterface
setDebugLevel
public static void setDebugLevel(int level)
Sets the current debug level. 0 means debug is not enabled.
Parameters:
level —the debug level
setModel
public void setModel(java.lang.String modelName)
Sets the model to be used by this Navigator. If the model has never been used before, tries to load the corresponding link descriptions from the repository and/or navigation bundle (the properties file modelName.propertiesmust be located directly under one of the directories in the class path or under a subdirectory called resources).
Parameters:
modelName —the model name descriptions associated to the given model
updateAllLinks
public int updateAllLinks(java.lang.Object object)
Invokes updateAllLinks(Object,Strinq,boolean) sending object.getClass( ).getName( ) as the map name and true as the updateRoot flag.
Parameters:
object —the object used as entry-point for navigating in the object graph
Returns:
the total number of objects updated, deleted or inserted
See Also:
updateAllLinks(Object,String,boolean)
updateAllLinks

public int updateAllLinks	(java.lang.Object obj, java.lang.String mapName, boolean updateRoot)

Updates the persistent data source with all changes made to memory graph image that are not yet propagated to the persistent graph image. If the updateRoot flag is true, the object given as parameter will be inserted (if it is not in the persistent graph image) or updated (if it is already in the persistent graph image).
All current and removed links will be followed recursively and marked for deletion, update or insertion, depending on if the link being followed was removed or not from the memory graph image. Deletion has priority over insertion and updates, which means that, if an objects is supposed to be deleted through a link and updated through another, that object will be marked for deletion. Note that deletion will cascade through a given link only if the object being marked for deletion is supposed to do so according to the current model. The same holds for insertion and update. After a successful invocation of this method, both memory and persistent graph images are synchronized.
If the class of the object given as parameter has a property (getter/setter pair or public field) with a name that matches a given link name, the referred objects reachable by that link will recursively "merge" into the object graph before the update phase takes place.
Parameters:
obj —the object used as entry-point for navigating in the object graph
mapName —the map name corresponding to the class of obj
updateRoot —indicates if obj must be inserted/updated
Returns:
the total number of objects updated, deleted or inserted
visitAllLinks

public int visitAllLinks	(java.lang.Object object, LinkVisitor visitor, boolean sync)

Invokes visitAllLinks (Object,String,LinkVisitor,boolean) sending object.getclass ( ).getName( ) as the map name.
Parameters:
object —the object used as entry-point for navigating in the object graph
visitor —LinkVisitor the visitor object
sync —boolean indicates if the memory image graph should be synchronized the persistent image in the end of the process

Returns:
int the number of objects visited
See Also:
visitAllLinks (Object,String,LinkVisitor,boolean)
clear
public void clear ( )
Clears the memory and persistent graph images held by this Navigator and resets all information about loaded links. This method is provided as a facility to re-initialize an instance, instead of having to create a new one. Note that the CocoBase connection and model associated to this Navigator remains unchanged.
getInstance
public static Navigator getInstance(java.lang.String instanceId)
Returns a Navigator object registered in the internal static pool. This method is provided as a facility to manage shared Navigator instances. Note, however, that this class is not thread safe and any necessary synchronizations must be properly implemented externally.
Parameters:
instanceId —the String that identifies the instance in the pool
getRegisteredIds
public static java.util.Enumeration getRegisteredIds( )
Returns an Enumeration object containing with all the ids of instances registered in the internal static pool. This method is provided as a facility to manage shared Navigator instances. Note, however, that this class is not thread safe and any necessary synchronizations must be properly implemented externally.
getTransaction
public Transaction getTransactions ( )
Returns the CocoBase transaction this Navigator should use to bind objects when updating the persistent data source.
Returns:
the CocoBase transaction; null if no transaction should be used
See Also:
setTransaction(thought.CocoBase.Transaction)
registerInstance

public static void registerInstance	(java.lang.String instanceId, Navigator nav)

Adds a Navigator object to the internal static pool. This method is provided as a facility to manage shared Navigator instances. Note, however, that this class is not thread safe and any necessary synchronizations must be properly implemented externally.
Parameters:
instanceId —the String that identifies the instance in the pool
nav —the Navigator instance to be registered
removeLinks
public void removeLinks(java.lang.Object obj)
Invokes removeLinks (Object, String) sending object.getclass( ) .getName( ) as the map name.
Parameters:
obj —the source object
See Also:
removeLinks (Object, String)
setTransaction
public void setTransaction(Transaction transaction)
Sets the transaction to which objects should be bound by this Navigator when updating the persistent data source. If this method is never called, this Navigator will check if there is a global transaction currently defined and the default behavior is to bind objects to that transaction. The default behavior can be overriden by calling setProperties(. . .).
Parameters:
transaction —the CocoBase transaction
See Also:
setProperties(java.util.Properties)
unregisterInstance
public static void unregisterInstance(java.lang.String instanceId)
Removes a Navigator object from the internal static pool. This method is provided as a facility to manage shared Navigator instances. Note, however, that is class is not thread safe and any necessary synchronizations must be properly implemented externally.
Parameters:
instanceId —the String that identifies the instance in the pool
findObject

public java.lang.Object findObject	(java.lang.Object object, java.lang.String mapName)

Finds the object instance in the memory graph image of this Navigator that is equal to the given object. The method equals is used for comparison.
Parameters:
object —the object to look for
Returns:
the object in this Navigator that is equal to the given object; null, if no instance is found
setProperties
public void setProperties(java.util.Properties properties)
Sets the properties that control the behavior of this Navigator. Currently supported properties are:
useCurrentTransaction ="true" (default) | "false"—a String indicating that this Navigator will try to bind objects to the global Transaction currently defined (when updating the persistent data source) in case no other transaction was assigned using the setTransaction (Transaction) method.
alwaysCreateProxies ="true" | "false" (default) —a String indicating that this Navigator will create link proxies for object properties (field or get/set method pair) even when the property is already populated (i.e. is not null).
Parameters:
properties —the properties to be assigned
See Also:
setTransaction(Transaction)

EXAMPLE 2

Company Example

A CD ROM is incorporated herein by reference to provide a list of the Java source code. properties definition and script files (text of the files) that may be utilized as a representative example with CocoNavigate. The CD incorporated by Reference is entitled Example 2 (and including a duplicate) included on a single disk with a single file titled Example 2, and sized at 370 KB.

The present invention may be embodied in specific forms other than those particularly described above or illustrated by the appended drawings. Upon viewing the present application preferred embodiments and other descriptions herein of the present invention, variations and other implementa-

tions that do not depart from the spirit and scope of the present invention will be apparent to one of routine skill in this field. Such variations and other implementations are considered part of the present invention and within the scope of the appended claims. Accordingly, reference should be made to the appended claims, rather than to the forgoing specification and drawings, as indicating the scope of the present invention.

We claim:

1. A local or distributed computer system for creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model, wherein the system comprises a user interface, a working memory storage area and at least one device for permanently storing information, and said system is loaded with at least one portion of a computer software program that comprises at least one user access interface and a set of programming routines designed for creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model and the said software program has at least one feature selected from the group consisting of:

a) a feature providing a method for displaying or printing a representation of a CDOG model, a definition representation or other representation thereof, as a graph or as a set of tables of complex data objects and relationships, and

b) a feature providing an interface for editing a CDOG model or for editing a definition or other representation thereof, or providing an interface for an editable input or source, such as a file, that can be modified to implement changes to a CDO or its relationships of a CDOG model.

2. A system according to claim **1**, providing an interface for displaying or editing the CDOG model, or for displaying or editing a definition or other representation of the CDOG model, as a graph or set of tables representing complex data objects and their relationships.

3. A method of using a local or distributed computer system for creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model, comprising the steps of:

a) loading at least one software program module on the local or distributed system for creating, maintaining, accessing, navigating and persisting objects as a CDOG model,

b) creating a CDOG representation definition,

c) monitoring and recording any changes to the CDOG on a local or distributed computer system, and

d) persisting any changes to the CDOG on a local or distributed computer system by updating the CDOG or CDOG representation definition to reflect and changes to any copies of the CDOG or CDOG representation definition, and saving a copy of the updated CDOG representation definition to a data source or to another type of information repository.

4. A method according to claim **3**, comprising the additional step of displaying or printing a representation of the CDOG model as a graph or as set of tables representing at least one complex object and a relationship to a data object, or a definition representation of at least one of either a complex data object and a relationship, or a definition representation of the complex data object and a relationship.

5. A computer software program for using a local or distributed computer system to create, maintain, access, navigate and persist complex data objects as a CDOG model, said software program comprises:

at least one user interface and at least one executable feature providing at least one action on a computer system selected from creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model,

wherein executing said program can create a new CDOG model, access a CDOG model stored from a tangible medium maintain a CDOG model, or persist a new or updated CDOG model to a storage medium.

6. A computer software program according to claim **5**, having at least one additional executable feature selected from the group consisting of:

a) a feature providing a method for displaying or printing a representation of a CDOG model, a definition representation or other representation thereof, as a graph or as a set of tables of complex data objects and relationships, and

b) a feature providing an interface for editing a CDOG model or for editing a definition or other representation thereof, or providing an interface for an editable input or source, such as a file, that can be modified to implement changes to a CDO or its relationships of a CDOG model.

7. An API that can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model, wherein the said API comprises at least one user access interface, at least one data source access interface and at least three main programming modules consisting of the following programming modules or routines:

I. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:

a) a computer programming sub-routine or sub-module for obtaining from a member selected from the group consisting of a data source, an information repository, and an input device, sufficient information to construct a CDOG model containing one or more CDOGs,

b) a computer programming sub-routine or sub-module for constructing and loading into the memory registers of the computer system a CDOG or a CDOG model representation definition from a data source or other repository, and

c) a computer programming sub-routine or sub-module for sending a copy of some portion or all of the CDOG representation definition to a user or to a software program module on a local computer system or on a distributed network;

II. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:

a) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system that has accessed or changed a portion of a CDOG or CDOG model, which is included in the CDOG, or CDOG model, representation definition of (I), above, and obtaining any changes to the CDOG or CDOG model,

d) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system who has obtained a copy of any portion of the CDOG, or CDOG model, repre-

sentation definition, and for obtaining a copy of any changes that the user might have made to any portion of the CDOG, or CDOG model, representation definition, and

e) a computer programming sub-routine or sub-module for comparing a copy of a CDOG, CDOG model, or a representation definition of either the CDOG or CDOG model, to an original stored version of the CDOG, CDOG model, or an original stored representation definition for the CDOG or CDOG model, and for updating the original to incorporate any changes to a CDOG or a representation definition that are made by the user or by a software program module; and

III. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or software program module of the computer system:

a) a computer programming sub-routine or sub-module for storing a new CDOG or CDOG model, or storing a definition of either the CDOG or CDOG model, in a data source or in another information repository, and

b) a computer programming sub-routine or sub-module for persisting (saving to permanent storage) either a changed portion of an updated CDOG, an updated CDOG model, or an updated definition representation for either a CDOG or a CDOG model, to a data source or to another type of information repository.

8. An API according to claim **7**, further having an interface that accesses and uses functionalities of an data source to data source mapping tool, associated programming routines, or associated class libraries.

9. An API according to claim **7**, further having an interface that accesses and uses functionalities of an object to relational or relational to object database mapping tool, associated programming routines, or associated class libraries.

10. An API according to claim **7**, having a feature for, or being associated with a computer program or module having a feature that provides a method for, displaying or printing a representation of a complex data object CDOG model, or for printing a definition or other defining representation thereof, as a graph or set of tables representing CDOs and their relationships.

11. A software tool comprising the API according to claim **7**, adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs throught network APIs.

12. A software tool according to claim **7**, adapted for performing the following steps:

a) reading a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file; and

b) converting the information of (a) into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file.

13. A software tool according to claim **12**, adapted for performing at least one of the following additional steps:

a) displaying a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

b) storing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

c) printing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

d) displaying the target information of (b);

e) storing the target information of (b); and

f) printing the target information of (b).

14. A software tool according to claim **13**, wherein the target information is at least one member selected from the group consisting a CocoBase database definition repository file, a CocoBase map, a CocoNavigate CDOG definition, a CoCoNavigate object link descriptor, and a CDOG object graph definition data file.

15. A software tool according to claim **13**, wherein the source is a UML/XMI document containing sufficient DTD information to exchange modeling information with a UML modeling tool.

16. A software tool according to claim **15**, wherein the software tool automatically generates a persistence layer that corresponds to the source UML class diagrams provided by an exported source file from a case tool capable of exporting UML/XMI such as Rational Rose, Together and ArgoUML.

* * * * *

EXHIBIT E

US007103600B2

(12) **United States Patent**
Mullins

(10) **Patent No.: US 7,103,600 B2**
(45) **Date of Patent: *Sep. 5, 2006**

(54) **DISPLAYABLE PRESENTATION PAGE AND SQL SEARCHABLE RELATIONAL DATA SOURCE IMPLEMENTATION OF A SYSTEM, METHOD AND SOFTWARE FOR CREATING OR MAINTAINING DISTRIBUTED TRANSPARENT PERSISTENCE OF COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS**

(75) Inventor: **Ward Mullins**, San Francisco, CA (US)

(73) Assignee: **Thought Inc.**, San Francisco, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 393 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/382,302**

(22) Filed: **Mar. 6, 2003**

(65) **Prior Publication Data**

US 2004/0177093 A1 Sep. 9, 2004

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/10**; 707/104.1

(58) **Field of Classification Search** ..... 707/1, 707/3, 10, 102, 103 R, 103 Y, 104.1; 705/1, 705/8; 709/219, 246; 713/200; 715/513
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,633,889 B1 * 10/2003 Dessloch et al. ..... 707/103 Y

* cited by examiner

Primary Examiner—Apu M. Mofiz
(74) *Attorney, Agent, or Firm*—Robert Lev

(57) **ABSTRACT**

The invention provides systems, methods and software for creating or maintaining distributed transparent persistence of complex data objects and associated data stores. In one aspect, the invention also relates to an application programming object capable of creating or maintaining distributed transparent persistence of data objects or data object graphs without the necessity of inserting any byte codes or modification of the object graph. Virtually any java object or object praph can be transparently persisted. Further, copies of a data graph or of a portion of the data graph can be automatically reconciled and changes persisted without any persistence coding in the object model.

17 Claims, 3 Drawing Sheets

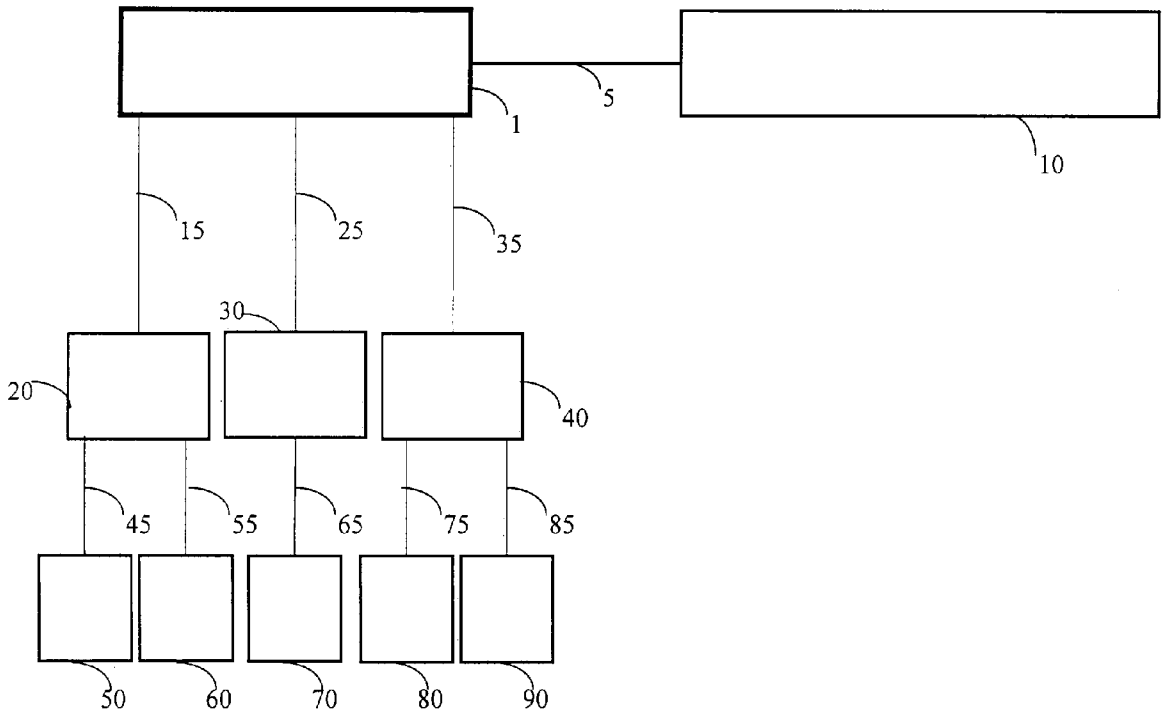


FIGURE 1

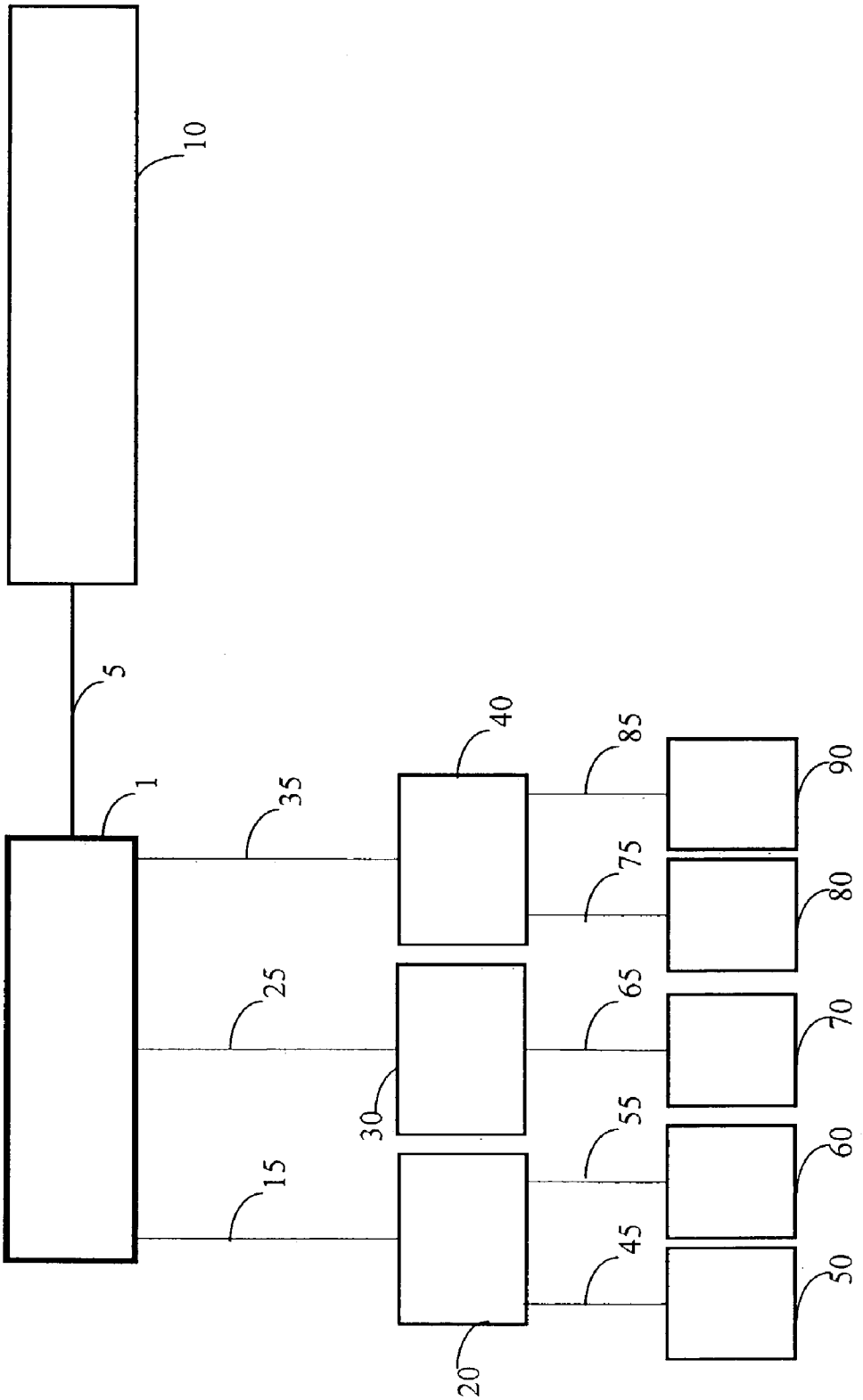


FIGURE 2

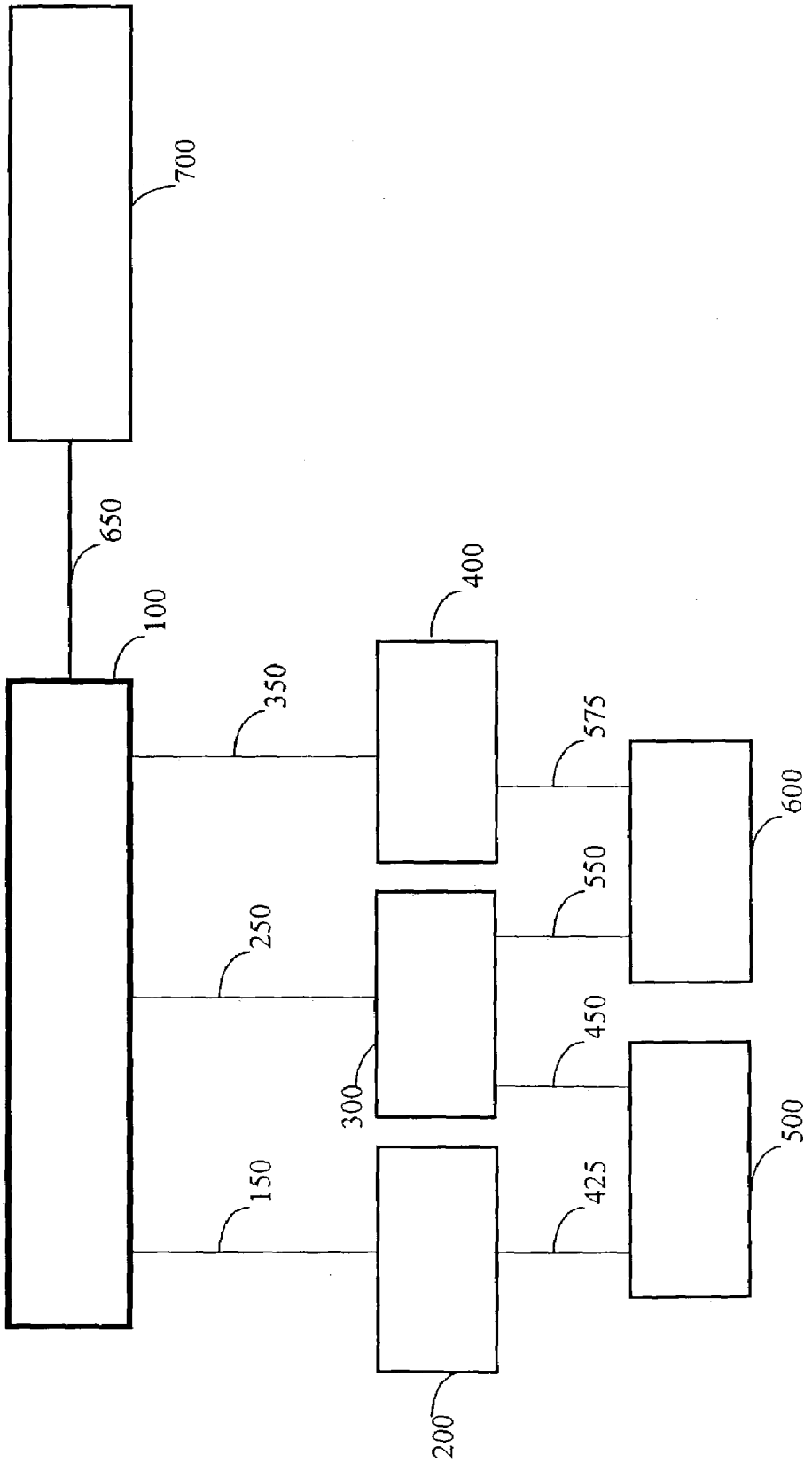
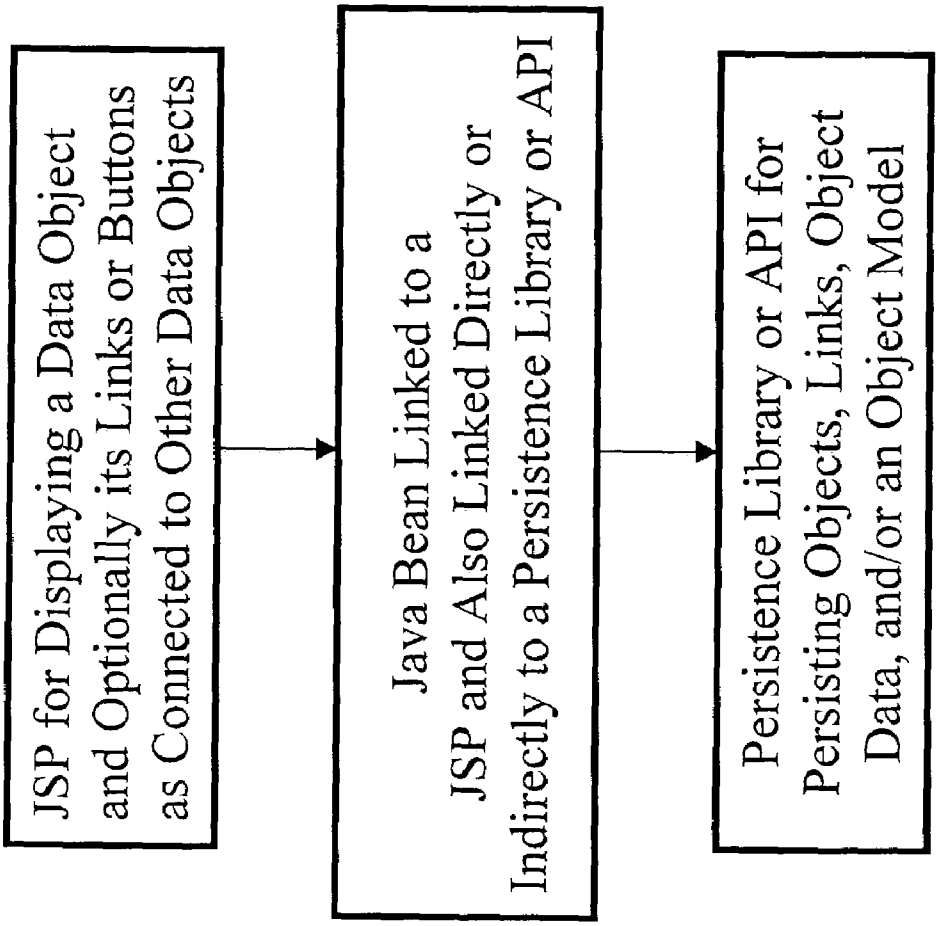


FIGURE 3


JSP for Displaying a Data Object and Optionally its Links or Buttons as Connected to Other Data Objects
Java Bean Linked to a JSP and Also Linked Directly or Indirectly to a Persistence Library or API
Persistence Library or API for Persisting Objects, Links, Object Data, and/or an Object Model

DISPLAYABLE PRESENTATION PAGE AND SQL SEARCHABLE RELATIONAL DATA SOURCE IMPLEMENTATION OF A SYSTEM, METHOD AND SOFTWARE FOR CREATING OR MAINTAINING DISTRIBUTED TRANSPARENT PERSISTENCE OF COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS

FIELD OF THE INVENTION

The field of the present invention relates generally to computer systems, computer data stores and to methods and software for accessing and utilizing data stores. More particularly, the present invention relates to a system, methods and software for creating or maintaining distributed transparent persistence of complex data objects and associated data stores. In one aspect, the invention also relates to an application programming object capable of creating or maintaining distributed transparent persistence of data objects or data object graphs without the necessity of inserting any byte codes or modification of the object graph. Virtually any java object or object graph can be transparently persisted. Further, copies of a data graph or of a portion of the data graph can be automatically reconciled and changes persisted without any persistence coding in the object model.

BACKGROUND OF THE INVENTION

Systems for accessing data stores from object oriented languages have been used for many years. A frequent approach to accomplish access of data stores involves writing and embedding custom access code within an object application needing the access. This approach is generally limited to having the custom code access only a single relational table within a relational database or similar construct within any other data store (hereinafter collectively "data store"). Under the circumstances where a developer has control over the design and creation of a data store from its inception, it is possible to design and store meaningful information in a single table. Such design opportunities are usually rare, however.

Generally, the methods for producing persistence for a data object, complex data object or a data store conflict with the goals of producing pure object application models where the object models do not include persistence objects or persistence byte code. Particular difficulties exist in a distributed environment since an object application model may exist in one or more of a computer's memory, an application data store and an application information storage repository that may be independent of the data store organization or object definitions. Advancements in the art have been made with respect to tools for conveniently mapping objects to systems of tables and maps in order to expedite accessing, changing and updating data stores. See, for example, U.S. Pat. No. 5,857,197 (and its associated programming interfaces ("APIs")) describes tools for translating object data to relational data, relational data to object data, and object data to object data to expedite the use of data stores. The BMP and the CMP Installer portions of CocoAdmin tool in the CocoBase™ Enterprise for O/R Binary Software (Thought, Inc. 657 Mission Street Suite 202, San Francisco, Calif. 94105 http://www.thoughtinc.com,) provide means for providing persistence in the EJB environment.

Persistence problems arise with the creation, access, changing or deleting of an object application model that utilizes such data stores. The object application model may

be distributed over multiple physical computer machine locations or even distributed over multiple Internet website locations that may be independent of the data stores. The object application model may utilize a different set of data objects or different set of definitions for relationships between data objects than that of one or more of its data sources. In most situations, the respective structures of the data sources and of the object applications model simply do not conveniently allow for mapping, accessing or changing of an overall schema of application data objects as well as any associated definitions of relationships between two or more data objects or elements within a data object.

Importantly, relationships may exist between a data object and one or more of the other data objects found in the object application model or in a data object of the data source. A relationship between one data object and another data object or with a data source may be member selected from the group of three relationship types consisting of 1 to 1 (1—1), 1 to many (1-M) or many to many (M—M). Complex combinations of these relationships may exist as a data object relationships definition for a given data object. These relationships are described or illustrated in further detail later in this document.

Objects may logically span multiple relational tables or multiple object databases, and may even be distributed over a logical (or hypothetical) computer system involving multiple physically independent computer systems or even multiple website locations. Creating, accessing, maintaining or updating an object application model can require working with multiple translation modules and require tedious and repetitive updating of multiple individual computer systems or multiple data sources in order to do useful work and keep the object application model synchronized. Such approaches are both costly and unwieldy in terms of computing and development resources, particularly with respect to Internet based electronic commerce (eCommerce) object application models.

Data objects of an object application model are often a feature of eCommerce object programming applications, where information is obtained from a data source and the data is defined as a data object (e.g., as a Java class) for use with another computer application. In practice, a data object or model of data objects may exist only in the random access memory of a computer memory system, or may be saved to either a data source or to some other type of retrievable information repository. A programmer or administrator of an object data application cannot easily access or track the overall model or diagram of data objects for an object application model or some of its specific elements. Unfortunately, tools for accessing and persisting data objects and associated data object relationships of a complex data object graph model have not been well implemented in the field of object language programming.

A computer application can execute one or more of the following non-limiting actions with respect to one or more of the members selected from the group consisting of data, a data object, and a data object definition: access data, change data, create data, create a new relationship between one or more data objects by creating or changing at least one data object relationship definition, change or delete a relationship between one or more data objects by changing or deleting at least one data object relationship definition, access a data object relationship definition and use its parameters to access a data source or a data object, and access one or more data object relationship definitions or data objects to create a new data object or data object relationship. Any changes executed by a computer applica-

tion with respect to one or more of the members selected from the group consisting of data, data object or data object definition may need to be properly persisted (permanently stored) to preserve any changes to one or more of the members selected from the group consisting of data, a data object and a data object definition.

A data object and an associated data object relationship definition may be represented by a complex data object graph ("CDOG"). A CDOG, for the purposes of this document, may be thought of as a computer program data object graph that represents a data object having at least one relationship with at least one other data object or with itself via a circular link. When the data object of a CDOG is implemented in the Java computer program language, the CDOG may be further defined as being a Java Data Object Graph ("JDOG").

There are needs for software, methods and systems that can easily detect and persist any changes to at least one member selected from the group consisting of a data object, any data associated with the related object, or any associated CDOG definition (i.e., an changes to the data object, data or to a relationship of the data object with another data object). For example, there is a need to be able access a pure object model definition from a repository based O/R mapping tool (e.g., from a defined repository such as a file, database or streaming source) or from a modeling tool defined repository and provide persistence for the object model without inserting any byte code or additional objects into the object model.

Accordingly, there is a strong need in the art for a computer applications programmer tool designed to assist a programmer or administrator in the actions of providing persistence for data objects or data object graphs when deleting, inactivating or updating a CDOG, wherein the computer applications programmer tool can be configured to automatically reconcile all or a portion of a CDOG and copies thereof on a distributed environment when data objects or relationships are deleted, inactivated or updated for a CDOG. A particularly strong need exists for such a tool having the further ability to be configured to persist, propagate and reflect system wide (in a local or distributed computer system) any such changes to a CDOG instance to all instances of the CDOG and to all instances of associated data, data objects and data object relationships.

DEFINITIONS

The following non-exhaustive list of definitions is used herein to define terms that may otherwise be confusing or can sometimes have multiple meanings. Each occurrence of a defined term in the above text, in the text that follows, or in the claims of this document, is to be given the meaning ascribed to it in the list of definitions below.

"Instance" as referred to in this document in the context of computer software applications is a single occurrence of a software logical element in the memory of a computer system, such as a "class", an "object", a "data object", and the like. This is analogous to the occurrence of a logical memory unit representing a row of data in common practice.

"Class" as referred to in this document in the context of computer software applications is a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). In practice, a class is a logical unit used as a logical template in an object oriented language from which to allocate new instances of objects.

"Object" as used in the context of this document is a general term referring to a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). The term "object" may ordinarily be used interchangeably with the term "class" as a template or as an instance depending on the context.

"Data object" as referred to in the context of this document represents the concept of the occurrence of an object that holds data within a specific computer application domain and is likely to have its contents stored in a persistent data source of a computer system (e.g., a database server, a binary file, a text file, or even in a combination of two or more of such a persistent data sources of a computer system). A data object may exist as an independent data object without any relationship to any other data object or it may have one or more relationships with itself or with one or more other data objects.

"Complex data object" (or "CDO") as used in the context of this document refers to the occurrence of a data object that has at least one or more relationships with itself, or at least one or more relationships with one or more other data object(s). In a given instance of a CDO at least one relationship is populated as a link, as defined below. A CDO may have a multiplicity of different relationships with itself or with one or more additional CDOs.

"Relationship" or "data relationship" as used in the context of a CDO refers to the type of logical combination that occurs between a data object with itself, or refers to the type of logical combination that occurs between a data object and at least one another data object. Among other references or descriptions, such a relationship is always referred to or partially described by a "relationship type". This term is used in an object oriented language context to reference or describe any expectations, actions and limitations possible between two or more data objects.

"Relationship type" in the context of this document is a label that specifies the possible multiple combinations that can occur between a CDO and itself or with at least one other CDO. The possible relationship type labels are 1—1 (one to one), 1-M (one to many) and M—M (many to many). A given CDO may be simultaneously related to more than one other CDO through several different types of relationship. This is also sometimes referred to as the multiplicity of the relationship.

"Link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself, between a CDO and another CDO. The occurrence of at least one populated link results in an instance of the CDO. This may be considered a synonym for a "relationship" between two objects.

"Circular link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself that may be direct or indirect (e.g., linked to itself through another CDO).

"Relationship definition" or "relationship description" in the context of this document and computer software applications refers to information, or an abstraction of information, regarding a "relationship", "data relationship" "relationship type" or a "link" that can be stored, accessed, transferred, communicated, displayed or edited.

"Complex data object graph" or "CDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships.

"Java data object graph" or "JDOG" is a term employed herein as an abstraction to logically represent a set of

complex data objects and a set of their corresponding relationships that are part of a Java programming application.

"Application model" or simply "model" are essentially interchangeable terms employed herein as abstractions to logically convey a collective description or other representation for a set of complex data objects and a corresponding description or other representation of their relationships. In one respect, these terms are used logically herein provide a general way of efficiently communicating when referring to set of metadata (i.e., data about data) that describes possible data entities (e.g., objects, database tables, maps, etc,) data relationship types, and data constraints involved in a computer system or application, or in a specific instance of an application. It is important to understand the context in which the terms "application model" and "model" are used in this document. Ordinarily computer engineers refer to the "model" as an abstraction rather than a specific possibility or instance of the model as applied. However, in this document for the ease of communication abstractions of the model, possible implementations of the model and instances of the model are all referred to generally as "application model" or "model". From the context of its use the term will be clear. The model is the abstract relationship between classes, wherein the CEDO is an instance (or set of instances) that express(es) the model.

"Navigation", "navigating" or "navigated" in the context of the present document refers to an action implementing at least one object to interact with a set of related objects for a certain purpose, such as creation, access, insertion, modification and deletion of an object, or of one of its relationships. It is the physical act of traversing the relationships, which might be referred to as "walking up or down the graph" in common lingo.

"Navigation model" as used herein is a special type of application model that is applied specifically to a description (or other representation) of how objects can relate to each other and what might be the expected behavior when a CDOG is navigated for a certain purpose.

"Object schema" is a term employed herein as an abstraction referring to the set of data object classes that describe the possible data objects that can be created, modified or maintained in an application, or describing an instance of a set of data object classes in an application.

"Distributed Transparent Persistence" is a term employed herein as an abstraction referring to the concept of providing persistence for a member selected from the group consisting of a data object, a data object graph, associated data and data object relationships in a distributed environment without the need for the insertion of byte code or data objects in an object model or schema.

"CocoBase Proxy Classes" is a term employed herein used in referring to wrapper classes that provide CocoBase runtime compatibility for objects that aren't inherently database aware. A computer system can persist the attributes and data for any data object that is wrapped with a CocoProxy wrapper class by simply using CocoBase facilities. For example, source code for the (attribute based) CocoProxy and (get/set method based) CocoProxyM classes are available under the thought\cocodemo3tier31\demos\pguide directory, when the CocoBase software tools suite is installed on a computer system. This optional design provides an extensible mechanism for instances of data extraction and propagation.

"CocoBase Navigation API" is a term employed herein to refer to an example of an API that provides database relationship mapping and object graph management capability for persistent objects. Database relationships are mapped to object links using CocoBase Navigator link definitions. Persistence control is provided at each class level in the object graph. Each of the Select, Insert, Update and Delete operations are individually configurable.

"CocoBase Transaction API" is a term employed herein to refer to an example of an API that provides object oriented transaction support. Transaction objects are used to persist data object attributes and maintain synchronization between database and in memory attribute values. The Transaction API has many built in optimizations, and applications utilizing CocoBase transactions generally benefit from reduced database and network overhead. This facility allows the developer to group together object changes into a single unit of work whose implementation or storage will succeed or fail all at once.

"CocoBase Factories" is a term employed herein to refer to examples of software modules and softwares libraries that are used to provide automated, custom object instantiation behavior. Factory behavior is completely customizable. For example, a factory may be used to bind newly instantiated objects to a transaction object, to load a graph of related objects using the CocoBase Navigator, or to implement polymorphism in a database result set. For example, a ProxyFactory class is part of the current CocoBase software tools suite distribution in the thought\cocodemo3tier31\demos\pguide directory, and this factory returns result set objects wrapped in a CocoProxy wrapper, when a CocoProxy wrapped key object is passed into the CocoBase runtime software module as part of a query that needs processing by the CocoBase runtime module.

"CocoBase Repository" is a term employed herein as an abstraction referring to a datasource to dataobject mapping repository and associated software modules that is installed into a datasource (or may optionally be a single stand alone file, or a set of files that circumscribe a set of datasource to dataobject mapping definitions and associated software modules). A repository can optionally be in a format such as XML, XMI and the like. See, U.S. Pat. No. 5,857,197, the CocoBaseEnterprise O/R Tools Suite, and the co-pending patent application entitled "Dynamic Object-Driven Database Manipulation and Mapping System" for more detailed descriptions of mapping repositories, and the like.

"CocoBase Transparent Persistence for Objects and Object Models". All models using a relational database for map storage require the CocoBase repository to be installed into the database, or in a stand-alone source accessable to CocoBase. The installation of a mapping repository can occur automatically, if required, when using CocoAdmin to log into the database. Pre-existing database tables can be used, provided that the CocoBase repository is first installed into the database, or accessible to CocoBase. Several example of applications that implement CocoBase transparent persistence are included in the CocoBase software tools suite distribution (see the distribution documentation for the location of folders or directories containing such examples).

SUMMARY OF THE INVENTION

An object of the present invention is to provide a system for creating or maintaining transparent persistence of a complex data object, a complex data object graph (CDOG) model, or a portion of a CDOG. In a preferred embodiment, an object of the present invention is to provide such a system that can selectively persist all or a portion of a CDOG model when the model is a member selected from the group

consisting of an object model generated from a data object mapping repository and an object model generated from data object modeling tool repository. A further object is to provide such a system is located on, or is part of, a local or distributed computer system.

An object of the present invention is to provide a method for creating, maintaining, accessing, navigating and persisting complex data objects stores in a repository. In a preferred embodiment, an object of the present invention is to provide such a method having the step utilizing the storage facilities of an enterprise EJB server to store and maintain the data object repository. In a further object, such a method involves a local or distributed computer system.

An object of the present invention is to provide a computer software component that operates in an EJB environment, or the like, wherein the component has the capacity to access an object model repository or an instance thereof in a computer memory or in another temportary computer storage store device and persist at least one action selected from the group consisting of creating, maintaining, accessing, navigating, updating or deleting complex data objects as a CDOG model. In a preferred aspect, the computer software component is an Enterprise Bean selected from the group consisting of Stateless, Stateful and Entity Beans. In a further preferred object the computer software component is an EJB Session Bean built on top of CocoBase runtime libraries having the ability to persist all or a portion of a CDOG model or instance thereof. An even more preferred object is to provide such a computer software component capable of transparently persisting all or a portion of a CDOG model or instance thereo for a local or distributed computer system and automatically reconciling and persisting any changes to an instance of the CDOG model or any changes to the repository definition for the CDOG model.

A preferred object of the present invention is to provide a software tool comprising the a navigation API and software components (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

A further object of the present invention is to a software tool capable of reading a source (or binary) programming object logic model or a database file (or other data source containing the stored object model information) in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file and converting the information into a target member selected from the group consisting of a database definition, XML mapping file, a database mapping definition file, and a CDOG definition file. In a preferred object, the software can automatically generate a transparent persistence layer that corresponds to the object model information of the source file.

A further object of the present invention is to provide a software module and source code known as a Java entity bean (such as a generic session bean) that is capable of providing persistence of either or both of a data objects and a data model, in total or in part as determined through setting established by a user of the computer system.

An object of the present invention is to provide a system and/or method for displaying, updating or creating the data or structure of a data object, and optionally its links to other objects of an object model, by utilizing a displayable presentation page having embedded object programming code, such as a Java Server Page (JSP) that is served up by a server platform such as a JSP server, and persisting data of the object, links of the object, or object model by communication of the object programming code embedded in the displayable presentation page with a programming object (such as a Java Bean) wherein the programming object has implemented in its object programming logic methods for getting, setting, resetting and loading of data, and the programming object also includes programming code for communicating directly or indirectly with a persistence manager API or persistence library to persist the object, its data and/or links to other objects in an object model, and optionally to persist the object model itself. In a preferred object of the present invention the displayable presentation page is constructed having an HTML format and the embedded programming language is the object programming language Java, and the associated programming object is a Java Object such as a Java Bean or entity bean.

Another object of the present invention is to provide a system and/or method for generating from a relational datasource schema, or generating from an Object to Relational mapping resource:

a. at least one displayable presentation page (such as a JSP) having object programming embedded within the page, with the presentation page having an ability to display data of an object in an object model, and optionally further display link information and/or to further provide a labeled link button for navigating to a linked object in an object model, and

b. generating at least one corresponding programming object, such as a Java Bean, for communicating with the generated presentation page and also for communicating with a persistence manager API that can persist the object, its data and/or links to other objects in an object model, and

c. optionally generating a set of related displayable presentation pages (such as JSPs) corresponding to linked objects of a specified object model.

d. optionally comprising an ability to query against the object graph using a object query facility or syntax.

Another object of the present invention is to provide a method of visually navigating, updating and persisting an object model by utilizing linked displayable presentation pages (such as JSPs) corresponding to a specified object model. A preferred object is to provide such displayable presentation pages with buttons that may be clicked to access link information or to move the user forward or backward to a navigated linked object of an object model and its displayable presentation page.

A further object of the present invention is to provide a method for creating, updating, navigating and persisting an object model starting from a information source which is member selected from the group consisting of Java Classes, a relational data source, a UML model (e.g., XMI respresentation), and interface for manually creating an object model by describing a set of objects, their attributes and their links to other objects, and then in a further step creating from the information retrieved from such an information source a set of displayable presentation pages and their associated programming objects (such as JSPs and associated Java Beans) corresponding to an object model that are suitable for visually creating, navigating or updating the object model.

BRIEF DESCRIPTION OF THE DRAWINGS

For the non-limiting purpose of illustrating some of the concepts of complex data objects CDOs, i.e., data objects

and their relationships to one another, according to the invention, two CDO graph drawings FIG. **1** and FIG. **2** are provided.

FIG. **1** is a complex data object (CDO) graph drawing, which illustrates a customer object and some of its related objects (billing address, orders and items ordered), as well as relationships between the objects. Relationships of the types 1 to 1 (1—1) and 1 to many (1-M) are shown in this CDO graph. More specifically, FIG. **1** illustrates a CDO graph drawing presenting an instance of a customer object **1** having a 1 to 1 (1—1) relationship (**5**) with its customer billing address object **10**, and a **1** to many relationship (collectively **15**, **25**, and **35**) with the three outstanding order objects **20**, **30** and **40**, respectively. Order object **20** is an instance of an outstanding order object having a 1 to many relationship (collectively **45** and **55**) with the two items ordered objects **50** and **60**, respectively. Order object **30** is an instance of an outstanding order object having a relationship with a single order item, but order object **30** has a 1 to many relationship (**65**) with the item ordered object **70**, since many order items could have been associated. Order object **40** is an instance illustrates a 1 to many relationship (collectively **75** and **85**) with the two items ordered objects **80** and **90**, respectively.

FIG. **2** is a complex data object (CDO) graph drawing, which illustrates a company object and some of its related objects (corporate address object and some of its departments and employees), as well as relationships between the objects. Relationships of all three types: 1 to 1 (1—1), 1 to many (1-M) and many to many (M—M) are shown in this CDO graph. More specifically, FIG. **2** illustrates a CDO graph drawing presenting an instance of a company object **100** having a 1 to 1 relationship (**650**) with its corporate address object **700**, and a 1 to many relationship (collectively **150**, **250**, and **350**) with the three company department objects **200**, **300** and **400**, respectively. Since employees of this company may work for more than one of the company's departments, the company department objects **200**, **300** and **400** in FIG. **2** are three instances (many) of company department objects having relationships (**425**, **450**, **550** and **575**, respectively) with two (many) employee objects (respectively, **500** and **600**). The cross-assignment of employee object **500** to both company department objects **200** and **300**, and of employee object **600** to both company department objects **300** and **400**, illustrate a complex many to many (M—M) relationship of departments to employees for this company.

FIG. **3** is a programming flow chart representing a preferred embodiment of the invention which is a system for creating, displaying, updating, and persisting the data and/or links of a complex data object (CDO) which is part of a CDO graph. The first programming module of the illustrated flow chart utilizes a displayable presentation page having embedded object programming code, such as a Java Server Page (JSP), wherein the displayable presentation page may be served up (presented) to a user by a server platform such as a JSP server. This displayable presentation page, among other things, may display the data of a data object and/or provide an interface for creating or updating the data for that data object (or for a new data object) and the displayable presentation page may also include one or more link names (or labelled link buttons) as connected to other data objects of the CDO graph/model. The embedded programming code of the displayable presentation page includes (i) logic for accessing data and and displaying the data from a particular data source and the logic may may perform this function directly or delegate it to at least one other programming module, (ii) logic for creating, updating or deleting data of an object, (iii) a programming reference link to an associated programming object (such as a Java Bean) that is, in turn, directly or indirectly connected to a programming module that contains additional logic for detecting changes to object data and/or changes to the object graph/model that includes the particular object or is associated with it, and (iv) logic for reflecting to the associated programming object the data and/or links to other objects of the data object that is being displayed by the presentation page. The second programming module of the illustrated programming flow chart is the programming object (such as a Java Bean) that is linked to the first programming module which is linked to at least one data object and is further linked to a third programming module (either directly or indirectly, such as through an API) which is a programming module and/or library that includes programming logic for for tracking and persisting to at least one data source changes to objects, object model links, object data and/or an object model. This second programming module contains programming logic (a) for issuing commands such as get, set, reset and load attributes of a dat object, (b) for detecting changes to one or more of an object, links between objects, or object model, (c) link to a third programming module for persisting the detected changes. The third programming module of the illustrated programming chart provides a persistence library or API with associated programming logic for tracking and persisting to at least one data source the changes to objects, object model links, object data and/or an object model associated with the displayable presentation page of the first programming module in the flow chart when it is notified of such changes by the second programming module. In one preferred embodiment the displayable presentation page includes buttons to links with related objects of an object model, the programming flow chart also represents the embodiment of a method for visually navigating a populated or unpopulated object model graph and/or its data and links between its objects.

DISCRIPTION THE INVENTION

The present invention provides a system for creating or maintaining persistence for all or a part of a complex data object graph model when a user is creating, maintaining, accessing and navigating complex data objects as a complex data object graph model. In one embodiment, the system comprises a computer system having a user interface, a working memory storage area and at least one device for permanently storing information, and said system being loaded with at least one portion of a computer software program, wherein said software program comprises at least one user access interface and a set of programming routines designed for creating or maintaining transparent persistence when a user is creating, maintaining, accessing and navigating complex data objects as a CDOG model, comprising

a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, wherein the set of definitions is stored in a repository;

b) a set of definitions for the relationships between objects for an object language application, wherein the set of definitions is part of an object application navigation model;

c) a list of objects, or a set of objects that are to be persisted, wherein the list of objects or set of objects is part of an object application navigation model;

d) an object or set of objects as a programming module that contains the logic for persisting an indicated object or set of objects from c);

e) an input method to inform the programming module of d) with the location of information corresponding to a), b) and c), and

f) at least one data source to store persisted data.

In a preferred embodiment, the present invention provides such a system that can persist any part or all of a CDOG model instance, and to resolve and persist any changes to the model or to the respository for the CDOG model. Most preferred is such a system providing a point and click graphical user interface.

The present invention provides a method for creating, maintaining, accessing, navigating and persisting complex data objects stores as a CDOG model and providing transparent persistence in a distributed environement, comprising the steps of:

a) creating or accessing a CDOG representation definition or accessing an instance of the CDOG,

b) monitoring and recording any changes to the CDOG or a portion thereof on a local or distributed computer system, and

c) persisting any changes to the CDOG on a local or distributed computer system by updating the CDOG or CDOG representation definition to reflect and changes to any copies of the CDOG or CDOG representation definition, and saving a copy of the updated CDOG representation definition to a data source or to another type of information repository.

In a preferred embodiment, the present invention provides such a method wherein the information repository is an object data store managed by an EJB Enterprise server. In a further preferred embodiment an EJB Enterprise Java Bean is the software component having the ability to access a CDOG repository file, a UML/XMI repository, or other mapping definition repository (e.g., file, database, or streaming data source) or an instance of the at least one portion of the CDOG in a computer's memory or on a distributed network and provide the transparent persistence for the CDOG model.

The present invention provides a computer software program having at least one user interface and having at least one feature that provides for at least one action selected from the group consisting of creating or maintaining transparent persistence when a user of a CDOG model is creating, maintaining, accessing or navigating a CDOG model. In a preferred aspect of the invention, the present invention provides a software program, or an association with a software program, having a feature for displaying, printing or displaying and printing a representation of the CDOG model as a graph, or as a set of tables representing a CDOG model. In a further preferred embodiment, such a software program has an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the complex data object CDOG model (or has both an interface for editing and an editable input or source, such as a file).

The present invention also provides an application programming interface ("API"), as a preferred embodiment, wherein the API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In one aspect the API can be accessed by an independent computer software program, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, the present invention provides such an API as part of a software tool package that provides a method for displaying or printing a representation of the CDOG model as a graph or set of tables representing the CDOG model (or providing methods for both displaying and printing a representation of the CDOG model). In another preferred aspect, such a software tool package that includes the API provides an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the CDOG model (or provides both an editing interface and an editable input or source, such as a file,).

A preferred embodiment of the present invention provides a software tool comprising the API according (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

In another preferred embodiment, the CDOG API is the CocoNavigator API, the Navigator API, or the CBFacade API, which provides support for manipulating complex objects in conjunction with the THOUGHT Inc. CocoBase Enterprise Object/Relational (O/R) database (hereinafter "CocoBase") mapping tool (see U.S. Pat. No. 5,857,197 (incorporated herein in its entirety), for concepts upon which the CocoBase mapping tool is based). An object is considered to be complex when it has established or potential relationships (i.e. links) to other objects. For instance, a Company object instance may be linked to an instance of its corporate address and instances of a Department object, which in turn may be linked to a set of instances of Employee objects. An example of such an CDOG is shown in FIG. **2**, for example. The combination of these objects and their links constitute a graph of objects (CDOG model) that can be manipulated using the CocoNavigator API. Since the CocoNavigator API works in a Java Programming language environment, this CDOG example may also be referred to as a JDOG example.

In one preferred embodiment, the invention provides a software module, or set of modules, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a CDOG model, comprising:

i) a software module for creating a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, wherein the software module is capable of causing the storage of the set of definitions in a repository;

ii) a software module for creating and accessing a set of definitions for the relationships between objects for an object language application, wherein the software module is capable of causing storage of the set of definitions in a navigation model;

iii) a software module providing an input means for a member selected from an object name, a list of objects, or a set of objects that are to be persisted, wherein the software module is capable of capable of causing the storage of the input as part of a navigation model that can be access by ii);

iv) a software module containing the programming logic to persist an object or set of objects based upon access to (a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, (b) a set of definitions for the relationships between objects for an object language appli-

cation, and (c) an object name, a list of objects, or a set of objects that are to be persisted, and

v) a software module having an input method capable of informing iv) of the location of information relating to iv)(a), iv)(b) and iv)(c).

In another preferred embodiment, the invention provides a software tool comprising the API according to the invention or interfacing therewith, wherein the software is adapted for performing the following steps:

a) reading or accessing an object model definition from a repository based O/R mapping tool (e.g., from a defined repository such as a file, database or streaming source) or from a modeling tool defined repository source in a format selected from the group consisting of a database schema, a source or binary object graph, Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, an Extensible Markup Language ("XML") file, and other general data formats; and

b) converting the information of (a) into a target member selected from the group consisting of a database defintion XML format, a database mapping definition format, other general data formats, a source or binary CDOG definition file, and application objects.

In a further preferred embodiment, the present invention provides a software tool as described above that is adapted for performing at least one of the following additional steps:

a) displaying an object model, definition or portion thereof in a format selected from the group consisting of a database schema, a source or binary object graph, Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, an Extensible Markup Language ("XML") file, and other general data formats;

b) storing a representation of a object model, defintion, or portion thereof in a repository (or temporary computer memory representation) in a format selected from the group consisting of a database schema, a source or binary object graph, Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, an Extensible Markup Language ("XMI") file, and other general data formats;

c) printing a representation of a object model, defintion, or portion thereof in a repository (or temporary computer memory representation) in a format selected from the group consisting of a database schema, a source or binary object graph, Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, an Extensible Markup Language ("XMI") file, and other general data formats;

d) displaying the target information of (b);

e) generating at least one application object from the information of (a) or (b); and

f) printing the target information of (b).

In a still further preferred embodiment the software tool described above provides target information that is at least one member selected from the group consisting a CocoBase database definition repository, a CocoBase map, a CocoNavigate CDOG definition, a CocoNavigate object link descriptor, and a CDOG object graph definition capable of being stored and retreived. Further preferred is such software, wherein the source is a UML/XMI document containing sufficient DTD information to exchange modeling information with a UML modeling tool.

In one embodiment of the software tool according to the invention, the software tool can be set to automatically generate a persistence layer that corresponds to the source UML class diagrams provided by an exported source file from a case tool capable of exporting UML/XMI, such as Rational Rose, Together and ArgoUML.

An easy way to understand the job performed by a CDOG Navigator API, such as the CocoNavigator API, is to imagine it as a monitor that can provide at least one object to intercept any accesses to the data objects and their relationships that constitute the CDOG, as well as any changes to the data source upon which the CDOG depends, in order to provide a means to persist any changes to the CDOG (or CDOG model) or related data in the data source. Any changes to the CDOG (or CDOG model) can then be propagated by the CDOG Navigator API to a persistent data source through a CocoBase Enterprise O/R connection. Similarly, persistent data that updates a data source can be utilized to create a CDOG model or to update a CDOG model.

Each CDOG (or CDOG model) managed by the CDOG Navigator API can be associated by the CDOG Navigator API with a CDOG descriptor (such as a file) that may be utilized to represent all or part of a "navigation model". In this respect, a navigation model may be abstractly described as essentially a namespace in which a set of data objects and a set of link descriptions (i.e., relationship types between data objects) are each defined. In a preferred embodiment, a data source, data field, data field size and data field type are listed for each data object in the set of data objects. In another preferred embodiment, at least one link description for two data objects, or for a single data object having a circular link (where the link description is utilized in conjunction with the CocoNavigator API and CocoBase mapping tool) contains one or more of the following types of information (however other related information may be utilized):

at least one link type (e.g., 1:1, 1:M, M:1 and M:M) between data objects

at least one link qualifier (currently supported qualifiers are STD and BEAN)

at least one associative CocoBase map (optional, for M:M links only)

the names of the CocoBase maps related to each of the data objects (usually two maps that are not necessarily distinct), and

information that may be obtained for each of the CocoBase maps related to the relationship between two data objects, including:

the name of a relationship link between the two data objects

the names of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects

the name of a data source for each of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects

the names of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)

the names of a data source for each of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)

the name(s) of any java classes corresponding to fields that would impact upon the relationship between two data objects

a setting parameter indicating whether the a loading of a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)

a setting parameter indicating whether a deletion of information from a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)

a setting parameter indicating whether an insertion of information into a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s), and

a setting parameter indicating whether an update of information in a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s).

Some preferred features provided by the CocoNavigator API and its associated software, when utilized with the CocoBase mapping tool, or with a similar mapping tool, are as follows:

(a) Provides Multiple Navigation Models for the Same CDOG Model

A preferred embodiment of the CocoNavigator API or an associated program module is configured to allow many navigation models to be used with the same set of java classes and CocoBase maps that are associated with a CDOG model. This preferred embodiment API, or an associated computer program module, may also be configured to permit a user to switch dynamically from one such navigation model to another. Thus, while a given application is being executed, it is possible to completely change the relationships between data objects and how CDOGs of a CDOG model should be managed.

(b) Circular Link Definitions

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage circular data object links. In a navigation model, a circular link may be defined as occurring when a data object is directly or indirectly linked to itself as part of a CDOG. Such relationships can be created, accessed, supported and correctly managed via the CocoNavigator API.

(c) Bi-Directional and Oriented Links

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage links between data objects as either a bi-directional link or as an oriented link. In this embodiment links between data objects are usually bi-directional links and can be navigated back and forth as an un-oriented navigational model of the CDOG model. With such a bi-directional link feature activated, any object in the navigation model in a given CDOG model can be used as a navigation entry point. In an oriented link navigation model, the link may be set as an oriented link accessible from a single direction. Thus, changes must be made from top down or from bottom up on a relationship tree representation of the navigation model, and some actions may need to originate from a pre-set node of the relationship tree. In this embodiment, such types of links can be created, accessed, supported and correctly managed via the CocoNavigator API.

(d) Dynamic Link Proxy Properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, is configured to permit a user to populate any object property (e.g., public field or a getter/setter pair) having its type declared as an object class with special dynamic proxies that can monitor and update the state of a relationship link. Examples of such types of object classes are thought.Cocobase.navapi.LinkObjProxy of CocoBase and subclasses of the Sun Microsystems java class, java.util.Vector or other similar collection classes. Bi-directional or single directional object references can be consistently maintained in this manner, or by other similar logical mechanisms.

(e) Automatic Synchronization of Object Properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to automatically merge a data object and its properties into a CDOG navigation model representation when the class of object being navigated (being created, accessed, supported or maintained) has a property (e.g., a field or a getter/setter pair) with a name matching the name of a corresponding link as defined in the navigation model.

(f) Pure Object Models

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to include a data object as a relationship link to another data object in a navigation model without the need to declare fields and properties for the data object. Such links should be limited however to cases where no computer code dependency will exist between the java classes that will be associated with such linked data objects of the navigation model. Thus, such a navigation model can be a pure abstraction, and more reusable than just a populated version or single hypothetical instance of the abstraction.

(g) Customized Link Visitors

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can provide an interface which can be configured to permit a visitor object to use this interface to visit (e.g., access or change) a data object that is part of a CDOG navigation model. In conjunction with CocoBase, an example of CocoNavigator API implementing this concept might using a LinkVisitor object (class). Other similar classes may be defined for this purpose. The visitor interface provides a way for a user to customize and adapt the software, and thereby allows a user to extend the functionality of the CocoNavigator API, or the functionality of an associated computer program module, to provide a desired customizable behavior in their CDOG navigation model.

(h) Distributed Environments

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to operate as a tool to create, access, support and correctly manage a CDOG navigation model in a server environment (e.g., in an EJB container) and to persist any changes to the CDOG navigation model when a navigation model is distributed across a local network or when the navigation model involves a distributed network (e.g., a navigation model distributed across internet connections). In one aspect, on the server side of the network, a CDO or a CDOG model of any complexity sent by clients (e.g., serialized copies) across a local network, or across a distributed network, can be correctly merged into a CDOG model by the CocoNavigator API, or by an associated computer program module. Additionally, the CocoNavigator API, or an associated computer program module, can be configured to send a CDO or CDOG model to a client along with link proxies serialized with parts of the

CDO or CDOG model that are being monitored by such link proxies. By sending the CDO or CDOG model copy along with such link proxies and associated parts of the CDOG or CDOG model to a client, dynamic link proxies (described above in (d)) can be used by the client side. Accordingly, a CDOG navigation model can be created, accessed, supported, managed and persisted over a distributed network.

(i) Group Loading or Lazy Loading of Links

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to monitor some or all of the data objects (and associated relationship definitions) participating in links of a CDOG navigation model. In a more preferred embodiment, the data objects (and associated relationship definitions) participating in links of a CDOG navigation model can be loaded as they are needed (lazy loading) from a persistent data source or from another type of information repository. This lazy loading feature can permit a very large CDOG navigation model to be loaded in a per-link basis as needed. In a more preferred embodiment, the lazy loading feature can be configured to prevent a link from being loaded twice and can be configured to detect cycles of related loads that have been previously loaded. Such configuration adaptability can be exploited to provide the more efficient use of the resources for a computer system or network or for a distributed computer network.

Some examples of preferred features that can be provided by the CocoNavigator API and its associated software, which are particularly enhanced when utilized with the currently existing CocoBase mapping tool, are as follows:

(aa) CocoBase Transactions

A preferred embodiment of the CocoNavigator API or an associated program module is configured to bind a data object or data object relationship of a CDOG navigation model with a CocoBase transaction. Such configurations may be utilized to optimize access to the database server and to enhance performance of the computer system, or systems, involved with the CDOG navigation model.

(bb) Non-CBProp Objects

The CocoNavigator API or an associated program module can be configured to automatically detect non-CBProp objects (objects with classes that do not implement the CocoBase persistence interface known as the CBProp interface), and automatically create proxies in order to persist such objects as part of a CDOG navigation model. This is based on the Java Bean naming patterns in the Java Language specification of Sun Microsystems, Inc.

(cc) Virtual Foreign Key Fields

The CocoNavigator API or an associated computer program module can be configured to use proxy classes of CocoBase, such as the CocoProxyM classes, when appropriate. Implementing proxy classes such as the CocoProxyM classes can provide a system for creating, accessing, supporting, properly maintaining and persisting virtual foreign key fields (i.e., foreign key fields do not need to be defined in the object class for Java programming implementations) by the CocoNavigator API, or by an associated computer program module.

(dd) Transparent Persistence

The CocoNavigator API or an associated computer program module can be configured to use a configurable network component capable of acquiring and persisting CDOGs through network APIs and thereby provide persistence transparently with respect to the applications that are using the CDOGs. In a preferred implementation, there is provided a software tool comprising the CocoNavigator API that is adapted for a local network or for a distributed network environment, wherein the software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGS. Such a concept may be referred to as transparent persistence in the context of this API, because persistence can be obtained without requiring reconfiguring of an existing CDOG model or software application in order to provide such persistence to a computer software application.

(ee) Transparent Persistence with Automated APIs that may be Further Optionally Utilized with a Local or Distributed Implementations Such as a Session Bean Implementation

The CBFacade or other modules containing high level API's can be utilized that can function in place of the CocoNavigator API (or co-ordinate with it) to automate many of the transparent persistence functions. Such a software module or modules can be utilized in concert with a local or distributed implementation such as a session bean to provide an easier and quicker establishment of transparent persistence. The CBFacade can provide the functions as defined in (dd) and the session bean can provide a communication link between modules such as a transaction API and the navigation API (and/or with the CBFacade) to assure that transparent persistence occurrs as required. Such transparent persistence in the context of this API can be obtained without requiring reconfiguring of an existing CDOG model or software application in order to provide such persistence to a computer software application.

From the above description of features of the API, and the features of the CocoNavigator API programming routines described below, an object computer language programmer can produce an API having the functionality of the CocoNavigator API described herein. Essentially the CocoNavigator API has an at least one user access interface, at least one data source access interface and at least three main programming modules consisting of the following programming modules or routines:

I. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:

a) a computer programming sub-routine or sub-module for obtaining from a member selected from the group consisting of a data source, an information repository, and an input device, sufficient information to construct a CDOG model containing one or more CDOGs,

b) a computer programming sub-routine or sub-module for constructing and loading into the memory registers of the computer system a CDOG or a CDOG model representation definition from a data source or other repository, and

c) a computer programming sub-routine or sub-module for sending a copy of some portion or all of the CDOG representation definition to a user or to a software program module on a local computer system or on a distributed network;

II. A programming module or routine constructed to operate on a computer system and to provide the

following features to a user or to a software program module of the computer system (i.e., a persistence manager)

a) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system that has accessed or changed a portion of a CDOG or CDOG model, which is included in the CDOG, or CDOG model, representation definition of (I), above, and obtaining any changes to the CDOG or CDOG model,

b) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system who has obtained a copy of any portion of the CDOG, or CDOG model, representation definition, and for obtaining a copy of any changes that the user might have made to any portion of the CDOG, or CDOG model, representation definition, and

c) a computer programming sub-routine or sub-module for comparing a copy of a CDOG, CDOG model, or a representation definition of either the CDOG or CDOG model, to an original stored version of the CDOG, CDOG model, or an original stored representation definition for the CDOG or CDOG model, and for updating the original to incorporate any changes to a CDOG or a representation definition that are made by the user or by a software program module; and

III. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or software program module of the computer system:

a) a computer programming sub-routine or sub-module for storing a new CDOG or CDOG model, or storing a definition of either the CDOG or CDOG model, in a data source or in another information repository, and

b) a computer programming sub-routine or sub-module for persisting (saving to permanent storage) either a changed portion of an updated CDOG, an updated CDOG model, or an updated definition representation for either a CDOG or a CDOG model, to a data source or to another type of information repository.

In a more preferred embodiment, the CDOG API according to the invention, (a preferred embodiment is the CocoNavigator API) can be written with an interface that accesses and uses functionalities of the CocoBase mapping tool, associated programming routines, or associated class libraries. Both the object code and the source code of the CDOG API are contemplated as part of this invention.

In another preferred embodiment, the above described CDOG API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In a particularly preferred aspect, the API can be accessed by a computer software program such as the CocoBase mapping tool, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In an even more preferred aspect, the present invention provides such an API as part of a software tool package having a feature or method for displaying or printing a graph or set of tables representing a CDOG or a CDOG model. In yet another preferred aspect, such a software tool package including the API provides an interface for editing a CDOG, a CDOG model, or for editing a representation definition for a CDOG or a CDOG model to provide a modification that implements a change to a CDOG, or to its definition.

As described above, a software component according to the invention that is capable of persisting all or a portion of a CDOG may be an Enterprise Java Bean selected from the group consisting of Stateless, Stateful or even Entity Beans. CocoBase Enterprise Beans (CBEBs) are a preferred embodiment of the invention and constitute a set of Enterprise Java Beans built upon or accessing CocoBase Runtime and EJB APIs to provide persistence in a generic, efficient and transparent way. Distributed transparent persistence can be provided without the need for any object model changes or byte code modifications. CBEBs can be directly deployed on an EJB server with or without customized settings and are capable of persisting virtually any Java Object, any Java Object graph or any portion of the Object graph. The Java Objects of any complexity that are arbitrarily created by client applications can be persisted, and the corresponding object classes can be properly mapped to a CocoBase repository or other object repository such as a UML XMI repository. Copies of the Java Object, any portion of an Object Graph, or all of an Object Graph can be automatically reconciled and the changes can be persisted to a storage repository.

The CocoBase Navigation API maps database foreign key-primary key relationships to object references. The relationship maps (sometimes referred to herein as navigation models), like the database maps, are stored in a special CocoBase repository, or may be converted into an XML file (or files) and stored in a stand-alone file or directory. The Navigation API supports 1 to 1, 1 to M, and M to M links in both the object and relational spaces. For M to M relationships, an associative table and a corresponding associative map are required, since relational databases do not directly support M to M relationships.

A navigation model must be created, or properly reference, before CocoBase links can be defined. A navigation model is a namespace used to categorize and store a set of link definitions. Generally, a single navigation model is sufficient for simple applications. Multiple navigation models are useful if an application switches between several relationship configurations. This allows different views of complex object data for the same set of underlying database tables. Having relationship mapping separated from table mapping is unique to CocoBase. Since maps can be used with multiple link models, it provides reusability at the mapping level.

Below is an example of how a client application on an EJB Server could connect to and use at least on CBEB to provide transparent persistence, even in a distributed environment.

EXAMPLE 1

```
// create a complex object having myObj as the root of the tree
. . .
Context initialContext = new InitialContext( );
SfCmbHome cmbHome =
(SfCmbHome)javax.rmi.PortableRemoteObject.narrow(
        initialContext.lookup("java:comp/env/ejb/sfcmbhome"),
SfCmbHome.class);
SfCmb myCmb = SfCmbHome.create(. . ./*connection
details*/. . .);
// create an example object
MyObject myObj = new MyObject( );
```

-continued

```
myObj.setSomeField(someValue);
// now ask the CMB to load the object from the database that
matches the example
myObj = myCmb.retrieveState(myObj);
myCmb.saveState(myObj);
. . .
// do some more changes to myComplexObject and save it
again
. . .
myCmb.saveState(myObj);
. . .
```

The above example can also be use to run a client application with the TomCat Server using a JSP to access a relational database (such as the Oracle Database) using at least CBEB to provide transparent persistenct, even in a distributed environment. Appendix 1 has more details of how this can be done.

Below is another example of how a client application on an EJB Server could connect to and use at least on CBEB to provide transparent persistence, even in a distributed environment. This example uses the CBFacade implementation.

EXAMPLE 2

```
// create a complex object having myObj as the root of the tree
. . .
CBFacade cbf =
CBFacade.create("thought.CocoBase.CBFacadeLocal");
Properties props = new Properties( );
props.put("cocosource.name",args[0]);
props.put("cocosource.jdbcdriver",args[1]);
props.put("cocosource.url",args[2]);
props.put("cocosource.user",args[3]);
props.put("cocosource.password",args[4]);
props.put("cocosource.navmodel","CompanyAppLinks");
cbf.connect(props);
// create an example object
MyObject myObj = new MyObject( );
myObj.setSomeField(someValue);
// now ask the CMB to load the object from the database
that matches the example
myObj = cbf.load(myObj);
cbf.save(myObj);
. . .
// do some more changes to myComplexObject and save it
again
. . .
cbf.save(myObj);
. . .
```

The present may be embodied in specific forms other than those particularly described above or illustrated by the appended drawings. Upon viewing the present application preferred embodiments and other descriptions herein of the present invention, variations and other implementations that do not depart from the spirit and scope of the present invention will be apparent to one of routine skill in this field. Such variations and other implementations are considered part of the present invention and within the scope of the appended claims. Accordingly, reference should be made to the appended claims, rather than to the forgoing specification and drawings, as indicating the scope of the present invention.

Appendix 1: A JSP Example Using the TomCat Server and a Relational Database and Providing Transparent Persistence using CocoBase Enterprise OR

I. Set up a relational database such as Oracle 8i and create a user called cityinfo.

II. Install TomCat as follows:

1. Install the TomCat Server 4.0.x by simply unpacking the binary distribution (.zip) into a convenient location.

On a Windows platform, open a command prompt and proceed as follows:

C> cd \

C> jar xf jakarta-tomcat-4.0.x.zip

On a Unix platform, open a command prompt and do something like:

/usr/local/bin% jar xf jakarta-tomcat-4.0.x.zip

2. Run the script jakarta-tomcat-4.0.x/bin/startup and open a browser on http://localhost:8080 to test the installation. The Jakarat Project Home page displaying the Tomcat logo will appear in your browser.
3. Run the script jakarta-tomcat-4.0.x/bin/shutdown to stop the server.

III. Configure the Environment as follows, while adjusting the environment to match a chosen relational database (this example uses Oracle8i as the relational database):

The following system variables are set for the applications to access the proper files and libraries. Set these variables through the Windows Control Panel under the System properties dialog. Example entries for these variables are listed below:

Variable	Entry	Description
JAVA_HOME	C:\jdk1.3.1	JDK home directory
PATH	C:\CINNT\System32;%JAVA_HOME%\bin	JDK utilities Windows executables
COCO_HOME	C:\thought\cocodemo3tier31	CocoBase home directory
CATALINA_HOME	C:\jakarta-tomcat-4.0.1	TomCat home directory
ORA_HOME	C:\Oracle\Ora81	Oracle home directory

IV. Installing CocoBase Enterprise O/R

Before installing CocoBase Enterprise O/R, confirm that it is a current release. The installation file will have the following naming format:

install_cocodemo3tierXX_XXXX.class

This is a fully functional copy of the software with a 30 day evaluation license. When license expires, the software will not function unless a commercial license is obtained and installed. In the directory containing the installation file, enter the following command:

java install_cocodemo3tier31_0516

The last four digits correspond to the software release date and may be different, than listed, for the latest release. If java.exe is not in the system PATH, an error will occur. If the environment is configured properly, the installer displays the initial CocoBase Intallation "Welcome" dialog.

The default settings should be selected during the installation. Fill in the user information fields completely to ensure the proper installation of the software. If the software will be used for commercial purposes, be sure to obtain a commercial license.

V. Configuring CocoBase Enterprise O/R for the Database

In this example, the database used is Oracle. Therefore, a specific User/Schema name must be specified in the Oracle.properties text file to view the details of that User/Schema in CocoAdmin (GUI interface for CocoBase). Locate and open the Oracle.properties file in the thought\cocodemo3tier31\demos\ resources directory. Change the SCHEMANAME entry from SCOTT to CITYINFO.

Edit the CocoAdmin.properties text file, in the same directory, to permanently change the login entries for CocoAdmin. Locate the Oracle Thin entries and make the following changes:

Change the COCOUSERID to CITYINFO

Change the COCOPASSWORD to CITYINFO.

Change the Oracle_Thin_COCODBURL entry to the name of the database host server (e.g. jdbc:oracle:thin:@localhost:1521:ORCL).

VI. Configuring the TomCat Server

The following steps describe how to configure the runtime classpath and an Oracle data source for the TomCat Server.

Setting up the Server Classpath

CocoBase runtime classes plus Oracle jdbc driver classes need to be included in the jvm process that runs TomCat. In order to do that, it is necessary to modify the script that launches TomCat as follows:

1. Open the script file %CATALINA_HOME%/bin/catalina.bat (Windows) or %CATALINA_HOME%/bin/catalina.sh (Unix) in your favorite text editor
2. Look for the "setup classpath" section and edit the settings of the environment variable CP to include cocobase runtime classes and resources and oracle jdbc driver classes as exemplified below:

   SET CP=%CP%; %COCO_HOME%\classes; %COCO_HOME%\demos; %ORA_HOME%\jdbc\lib\classes12.zip

Another alternative is to copy %COCO_HOME%\demos\CocoDeploy.jar (to generate it, use the script %COCO_HOME%/demos/mkcocodirs) and other required Java archives (e.g. oracle jdbc driver classes) to %CATALINA_HOME%/common/lib, but note that these files need to terminate with .jar (.zip files need to be renamed to jar).

Creating a DataSource for the Cityinfo Application

An Oracle jndi datasource properly configured is required for the TomCat server. All datasource configuration is stored in the file %CATALINA_HOME%/conf/server.xml. The following steps describe how to add a datasource entry to this configuration file.

1. Open the configuration file %CATALINA_HOME%/conf/server.xml in your favorite text editor

2. Edit the <Host> section by adding a <DefaultContext> entry like the following:

```
. . .
  <Host. . .>
  . . .
    <DefaultContext>
      <Resource name="jdbc/MyObjDataSource" auth="Container"
                type="javax.sql.DataSource"/>
      <ResourceParams name="jdbc/MyObjDataSource">
        <parameter><name>user</name><value>cityinfo</value>
        </parameter>
        <parameter><name>password</name><value>cityinfo</value>
        </parameter>
        <parameter><name>driverClassName</name>
          <value>oracle.jdbc.driver.OracleDriver</value></parameter>
        <parameter><name>driverName</name>
          <value>jdbc:oracle:thin:@localhost:1521:ORCL</value>
          </parameter>
      </ResourceParams>
    </DefaultContext>
  </Host>
. . .
```

3. If the server is already running, make sure it is restarted.

VII. Generating the TomCat Application Object Code with CocoBase Enterprise O/R

Standard code generation with CocoAdmin of CocoBase Enterprise O/R is utilized to produce the JSP for TomCat, which procedures are generally known in the art as described in prior publications by Thought, Inc regarding CocoBase Enterprise O/R. However, at the Code Generation Template pull-down menu of CocoAdmin the user need to select JSP for Entity Bean from the pull-down Java Code generation menu list. The other general procedures of CocoAdmin are followed to produce the generic web application (WAR file).

VIII. Deploying the Generated Web Application (WAR file) in TomCat

1. Copy the generated war file to %CATALINA_HOME%/webapps
2. Start the TomCat server by opening a command prompt and running the script %CATALINA_HOME%\bin\startup. This will expand the .war file under the %CATALINA_HOME%\webapps directory.

Notes:

At startup, check if the CLASSPATH entry includes all CocoBase and Oracle jdbc driver classes to make sure the startup script was properly edited.

To redeploy the war file, make sure you delete the old war file along with the whole expanded directory structure

under the %CATALINA_HOME%\webapps before copying the new war file; then, restart the server.

IX. Viewing the Deployed Generated Web Application (Cityinfo JSP)

The user is now ready to connect to the Cityinfo JSP. Be sure that the TomCat Server has been started. Open a web browser and enter the following URL:

http://localhost:8080/Cityinfo/Cityinfo

A webpage is displayed with a title of "CityInfo Generated Default JSP", or the like. The web page contains entry fields of "Name" (* indicates that this is a primary key field), "Country" and "Population" and allows the actions of "Insert", "Update", "Delete" and "Find" for such entries.

Any entries made, updated or deleted in this JSP are automatically persisted to the relational data source by CocoBase.

The three primary source code listing for the above application are set forth below.

Displayable Presentation Page (Cityinfo JSP) Example Written in HTML Format With Embedded Java Code

This Cityinfo JSP is an example of a displayable presentation page that has embedded within the presentation page mark-up language some necessary object programming language logic. CocoAdmin automatically generates the HTML and embedded Java code of this JSP (displayable presentation page) along with the Java Code for a First Java Object (JSP Processor Object), and a Second Java Object (Business Object) from an object to relational schema map corresponding to a relational data source in the above example. However, these same object code listings could have been hard coded by hand by a competent programmer and/or produced from information taken from another source such a UML/XMI object model, directly from a relational data source, or from the database schema for a data source.

The presentation mark-up language (in this example HTML) provides a WEB page (presentation page) to display the names of three attributes of a Cityinfo Object "Name", "Country", and "Population" (may be referred to as fields in a corresponding data source) with 4 possible actions "Insert", "Update", "Delete" and "Find" and a button labeled "Submit" for executing one of the four possible commands (actions). The embedded Java code in this Cityinfo JSP example communicates directly with the First Java Object (the object is a Java Bean in this example, which may be referred to as a JSP Processor Class) and also references the name of the Java Bean (JSP Processor Class) having the logic for carrying out or delegating the "insert/create", "update", "delete", "select/find" and "submit" commands of the JSP and logic for tracking the values and properties of the three named attributes/fields in this example.

```
<html>
<jsp:useBean id="CityinfoProcessor" scope="session"
class="Cityinfo.CityinfoJspProcessor" />
<jsp:setProperty name="CityinfoProcessor" property="*" />
<>
<>
<html>
<head>
```

-continued

```
<title>Cityinfo JSP</title>
</head>
<body background="back.gif">
<font size = 5 color="#CC0000">
<h1><b><center>Cityinfo JSP Example</center></b></h1>
<hr>
<br>
<form method=POST action=Cityinfo.jsp>
<BR>
Enter Values to lookup a specific Object.
<BR>
<br>Name <INPUT type=text name="name" size="8"
value="<jsp:getProperty
name="CityinfoProcessor" property="name" />" >
<br><br>Country <INPUT
type=text name="country" size="8" value="<jsp:getProperty
name="CityinfoProcessor" property="country" />" ><br><br>Population
<INPUT type=text name="population" size="8" value="<jsp:getProperty
name="CityinfoProcessor" property="population" />" ><br><br>
<h2><b>Action :</b></h2>
<INPUT type="radio" name="action" value="create">Insert
<INPUT type="radio" name="action" value="update">Update
<INPUT type="radio" name="action" value="delete">Delete
<INPUT type="radio" name="action" value="select">Find
<br>
<INPUT type=submit name="submit" value="Submit">
</form>
</FONT>
</body>
</html>
<hr>
<h3><b>Status :</b></h3> <>
</html>
```

Cityinfo First Object Java Code Example

This CityinfoJspProcessor Object is a First Java Object (in this case a Java Bean) that CocoAdmin generates along with the JSP (displayable presentation page) and the Second Java Object. CocoAdmin generates this JSP Processor Java code and compiled Java Class from an object to relational schema map corresponding to a relational data source, however these same object code listings could have been hard coded by hand and/or produced from another information source UML/XMI or directly from the database schema. Further, the Java Code of the First and Second Objects could have been consolidated into a single object, but this would have violated the specification for Java and JSPs and also reduced the programming flexibility of the system. By placing the more complex logic in the business object (second object) and keeping it separate from the JSP Processor Object (first object), the JSP and first object can be utilized by multiple applications.

This First Java Object is a Java Bean that references and represents the attributes of its corresponding JSP (displayable presentation page) and includes programming logic to execute the commands (or to delegate the execution of such commands) of the JSP, as described above. Since using standard JSP standards and Java Procedures would only permit the handling of a limited number of data types, this First Java Object references the Second Java Object (Cityinfo Object) which also reflects the data and data types (attributes of the First Java Object) but includes more business logic to convert forward/reverse data types at the discretion of the programmer by merely editing the business logic. An example of the programming code for the First Java Object is set forth below:

```
package Cityinfo;
import java.sql.*;
import java.io.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.transaction.UserTransaction;
import java.util.*;
import java.math.*;
/**
* EJB Jsp Processor Class Generated by CocoAdmin from THOUGHT
Inc.
*
* Copyright (c) 1995–2000 THOUGHT Inc. All Rights Reserved.
**/
public class CityinfoJspProcessor implements java.io.Serializable
{
private String action = null;
private CityinfoHome cityinfoHome;
private UserTransaction utx = null;
public CityinfoJspProcessor( ) {
    try {
        Properties env = System.getProperties( );
        Context ic = new InitialContext(env);
        try {
        utx = (UserTransaction)
ic.lookup("javax.transaction.UserTransaction");
        } catch (Exception e) {
        System.err.println ("Couldn't Create User Transaction!!!");
        e.printStackTrace( );
        }
        cityinfoHome = (CityinfoHome) ic.lookup("CityinfoApp");
    } catch (Exception re) {
        System.err.println ("Couldn't locate Cityinfo Home");
        re.printStackTrace( );
    }
    reset( );
}
public String getAction( ) {
    return action;
}
public void setAction(String a) {
System.out.println("Setting action : " + a);
    action = a;
}
    /**
     * Get the attribute name value.
     * @return name Returns the database attribute value is current set
to.
     **/
    public String getName( )
    {
        return name;
    }
    /**
     * Get the attribute country value.
     * @return country Returns the database attribute value is current
set to.
     **/
    public String getCountry( )
    {
        return country;
    }
    /**
     * Get the attribute population value.
     * @return population Returns the database attribute value is
current set to.
     **/
    public Integer getPopulation( )
    {
            return population;
    }
    /**
     * Set the attribute name value.
     * @param n__name the database attribute value to set the variable
to.
     **/
    public void setName(String n__name)
```

-continued

```
    {
        name = n__name;
    }
    /**
     * Set the attribute country value.
     * @param n__country the database attribute value to set the variable
to.
     **/
    public void setCountry(String n__country)
    {
        country = n__country;
    }
    /**
     * Set the attribute population value.
     * @param n__population the database attribute value to set the
variable to.
     **/
    public void setPopulation(Integer n__population)
    {
        population = n__population;
    }
    private String name = null;
    private String country = null;
    private Integer population = null;
    /**
     * Check to see if values are equal - this should be based on key
fields.
     *
     * @param n__population the primary key instance to compare to.
     **/
    public boolean equals(Object t__CityinfoJspProcessor)
    {
        if (! (t__CityinfoJspProcessor instanceof
CityinfoJspProcessor))
        {
            return false;
        }
        CityinfoJspProcessor t__CompareCityinfoJspProcessor =
(CityinfoJspProcessor)t__CityinfoJspProcessor;
        if(t__CompareCityinfoJspProcessor.name == null)
        {
            if(name != null)
            {
                return false;
            }
        }
        else if(!t__CompareCityinfoJspProcessor.name.equals(name))
        {
            return false;
        }
        return true;
    }
private void reset( ) {
final String emptyString = "";
final Integer zeroInteger = new Integer(0);
final BigDecimal zeroBigDecimal = new BigDecimal(0);
final Short zeroShort = new Short((short)0);
final Long zeroLong = new Long((long)0);
final Timestamp zeroTimestamp = new Timestamp((long)0);
final Time zeroTime = new Time((long)0);
final Float zeroFloat = new Float((float)0);
final Double zeroDouble = new Double((double)0);
final byte emptyByte[] = new byte[0];
final java.sql.Date zeroDate = new java.sql.Date((long)0);
setName(emptyString);
setCountry(emptyString);
setPopulation(zeroInteger);
}
public String toString( ) {
StringBuffer output = new StringBuffer( );
        output.append(" name="+getName( ));
        output.append(" country="+getCountry( ));
        output.append(" population="+getPopulation( ));
return output.toString( );
}
//
// Perform the action requested by user. Return status.
//
```

-continued

```
public String processRequest( ) {
    CityinfoInterface cityinfo = null;
    String message = "";
    System.out.println("Process request called ");
    System.out.println(this);
    try {
      if( action.equals("create") ) {
      CityinfoPK cityinfopk = new CityinfoPK( );
    try {
      utx.begin( );
        cityinfopk.name = getName( );
        cityinfo = cityinfoHome.findByPrimaryKey( cityinfopk );
        setCountry( cityinfo.getCountry( ) );
        setPopulation( cityinfo.getPopulation( ) );
        if(cityinfo != null) {
          message = "Entity bean "+cityinfopk+" already exists!
Duplicate Bean cannot be created!";
          utx.commit( );
          return message;
        }
    } catch (Throwable t) {
//t.printStackTrace( );
      //utx.rollback( );
    }
    try {
      //utx.begin( );
        cityinfoHome.create( getName( ), getCountry( ),
        getPopulation( )
);
      message = "Created Cityinfo "" + cityinfopk + """;
      utx.commit( );
    } catch (Exception e) {
e.printStackTrace( );
        message = "Exception on Cityinfo Create ="+e;
    }
    }
    else if( action.equals("update") ) {
    try {
      utx.begin( );
       CityinfoPK cityinfopk = new CityinfoPK( );
       cityinfopk.name = getName( );
       cityinfo = cityinfoHome.findByPrimaryKey( cityinfopk );
       setCountry( cityinfo.getCountry( ) );
       setPopulation( cityinfo.getPopulation( ) );
       message = "Set Cityinfo " + this ;
      utx.commit( );
    } catch (Exception e) {
        message = "Exception on Cityinfo Update="+e;
    }
    }
    else if( action.equals("select") ) {
    try {
        CityinfoPK cityinfopk = new CityinfoPK( );
        cityinfopk.name = getName( );
        cityinfo = cityinfoHome.findByPrimaryKey( cityinfopk );
        setCountry( cityinfo.getCountry( ) );
        setPopulation( cityinfo.getPopulation( ) );
        message = "Found Cityinfo " + this;
    } catch (Exception e) {
        message = "Exception on Cityinfo Select="+e;
    }
    }
    else if( action.equals("delete") ) {
    try {
      utx.begin( );
       CityinfoPK cityinfopk = new CityinfoPK( );
       cityinfopk.name = getName( );
       cityinfo = cityinfoHome.findByPrimaryKey( cityinfopk );
       cityinfo.remove( );
       message = "Removed cityinfo " + cityinfopk;
       cityinfo = null;
      utx.commit( );
    } catch (Exception e) {
        message = "Exception on Cityinfo Delete="+e;
    }
    }
```

-continued

```
    if( cityinfo != null ) {
    }
    else {
      reset( );
    }
  }
  catch (Throwable e) {
    message = e.toString( );
  }
    return message;
}
}
```

Cityinfo Second Object Java Code Example

This Cityinfo Object is a Second Java Object that CocoAdmin generates along with the JSP (displayable presentation page) and the First Java Object. CocoAdmin generates the HTML with inbedded object code (JSP), a First Java Object (JSP Processor Object), and this Second Java Object (Business Object) from an object to relational schema map corresponding to a relational data source, however these same object code listings could have been hard coded by hand and/or produced from another information source UML/XMI or directly from the database schema. Further, the Java Code of the First and Second Objects could have been consolidated into a single object, but this would have violated the specification for Java and JSPs and also reduced the programming flexibility of the system. By placing the more complex logic in the business object (second object) and keeping it separate from the JSP Processor Object (first object), the JSP and first object can be utilized by multiple applications.

This Second Java Object (Java object in this case) is a separate business object that sits in the programming logic flow chart between the First Java Object (JSP Processor Class) and the persistence library or persistence API. This Second Java Object reflects the data of the first Java Object (JSP Processor Class) and allows for further processing to create additional data types from the reflected data types and to translate backwards to the more limited reflected data types that are displayed by the JSP. This provides more complete support for all Java Data Types that might need to be properly retrieved from a data source and persisted back to that or another data source (the Java JSP Processor specification as established by SUN Microsystems, Inc. only supports a small, more limited subset of Java data types). This Second Java Object processes commands passed to it by the JSP via the First Java Object and automatically translates and reverse translates the data types passing through it to the appropriate data types for displaying or persisting. This further processing allows more complex data types and business logic to be communicated with the persistence library or persistence API than would be possible if the JSP Processor Class communicated directly with the persistence library or API (unless the JSP specification is violated by editing the JSP Processor Class source code (First Java Object Code) to include additional logic within the JSP Processor Class). An example of the Java Code for this Second Java Object, i.e., the Cityinfo Object (business object) is set forth below:

```
package Cityinfo;
import thought.CocoBase.*;
import java.math.*;
import java.sql.*;
import java.util.*;
import java.io.*;
import java.lang.*;
/*
* Generated by GenCBClass from THOUGHT Inc.
*
* Copyright (c) 1995,2000 THOUGHT Inc. All Rights Reserved.
*
*/
public class Cityinfo extends Object implements
Cloneable,java.io.Serializable,CBProp,thought.CocoBase.EJBPrimaryKey
{
    /**
     * Constructor for Cityinfo
     */
    public Cityinfo( )
    {
    }
    /**
     * Get the attribute name value.
     * @return name Returns the database attribute value is current set
to.
     */
    public String getName( )
    {
        return name;
    }
    /**
     * Get the attribute country value.
     * @return country Returns the database attribute value is current
set to.
     */
    public String getCountry( )
    {
        return country;
    }
    /**
     * Get the attribute population value.
     * @return population Returns the database attribute value is
current set to.
     */
    public Integer getPopulation( )
    {
            return population;
    }
    /**
     * Set the attribute name value.
     * @param n_name the database attribute value to set the variable
to.
     */
    public void setName(String n_name)
    {
        name= n_name;
        if(name == null) nameNullIndicator = true;
        else
            nameNullIndicator = false;
    }
    /**
     * Set our Query Condition for column name for dynamic querying.
     */
    public void modifyNameMatch(int t_match) {
        nameMatchCondition = t_match;
    }
    /**
     * Get our Query Condition for column name for dynamic querying.
     */
    public int retrieveNameMatch( ) {
        return nameMatchCondition;
    }
    /**
     * Set the attribute country value.
     * @param n_country the database attribute value to set the variable
to.
     */
    public void setCountry(String n_country)
```

-continued

```
    {
        country= n_country;
        if(country == null) countryNullIndicator = true;
        else
            countryNullIndicator = false;
    }
    /**
     * Set our Query Condition for column country for dynamic
      querying.
     */
    public void modifyCountryMatch(int t_match) {
        countryMatchCondition = t_match;
    }
    /**
     * Get our Query Condition for column country for dynamic
      querying.
     */
    public int retrieveCountryMatch( ) {
        return countryMatchCondition;
    }
    /**
     * Set the attribute population value.
     * @param n_population the database attribute value to set the
variable to.
     */
    public void setPopulation(Integer n_population)
    {
        population= n_population;
        if(population == null) populationNullIndicator = true;
        else
            populationNullIndicator = false;
    }
    /**
     * Set our Query Condition for column population for dynamic
querying.
     */
    public void modifyPopulationMatch(int t_match) {
        populationMatchCondition = t_match;
    }
    /**
     * Get our Query Condition for column population for dynamic
querying.
     */
    public int retrievePopulationMatch( ) {
        return populationMatchCondition;
    }
    /**
     * Implement clone so update method works correctly
     */
    public Object clone( )
    {
        Cityinfo retObject = new Cityinfo( );
        retObject.setName(name);
        retObject.setCountry(country);
        retObject.setPopulation(getPopulation( ));
        return retObject;
    }
    /**
     * Implement toString( ) to properly print contents
     */
    public String toString( )
    {
        String strout =
        " name="+name+
        " country="+country+
        " population="+population+
        "\n";
        return strout;
    }
    /**
     * Take Properties result set received from database, and populate
     * Object's Data fields.
     * @param resultSet The data used in populating the Object's Data
fields
     */
    public void setPropObjectData(Properties resultSet)
    {
        if(resultSet.get("NAME") != null && !(resultSet.get("NAME")
instanceof CBNull))
```

-continued

```
                name = (String)resultSet.get("NAME");
            else {
                name = null;
                nameNullIndicator = true;
            }
            if(resultSet.get("COUNTRY") != null &&
!(resultSet.get("COUNTRY") instanceof CBNull))
                country = (String) resultSet.get("COUNTRY");
            else {
                country = null;
                countryNullIndicator = true;
            }
            if(resultSet.get("POPULATION") != null &&
!(resultSet.get("POPULATION") instanceof CBNull))
                population = ((Integer)resultSet.get("POPULATION"));
            else {
                population = null;
                populationNullIndicator = true;
            }
        }
        /**
         * Take the data in the Object's Data fields, and create a
CBQueryInfo
         * instance, which can be used by CocoBase. Since fields are
identified
         * by a key value defined by their name, field order is irrelevant
with
         * this interface.
         * @return CBQueryInfo the Object's Data fields and query info
         * @see CBQueryInfo
         */
        public CBQueryInfo getPropObjectData( )
        {
            cqi = new CBQueryInfo( );
            setProps = new Properties( );
            if(name != null) {
                // Use the CBFieldInfo to specify the match condition
                if(nameMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
                    cbfi.setSearchValue(name);
                    cbfi.setSearchCriteria(nameMatchCondition);
                    setProps.put("NAME" ,cbfi);
                } else
                    setProps.put("NAME",name);
            }
            else if(nameNullIndicator) {
                // Use the CBFieldInfo to specify the match condition
                if(nameMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
                    cbfi.setSearchValue(name);
                    cbfi.setSearchCriteria(nameMatchCondition);
                    setProps.put("NAME",new CBNull( ));
                } else
                    setProps.put("NAME",new CBNull( ));
            }
            if(country != null) {
                // Use the CBFieldInfo to specify the match condition
                if(countryMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
                    cbfi.setSearchValue(country);
                    cbfi.setSearchCriteria(countryMatchCondition);
                    setProps.put("COUNTRY",cbfi);
                } else
                    setProps.put("COUNTRY",country);
            }
            else if(countryNullIndicator) {
                // Use the CBFieldInfo to specify the match condition
                if(countryMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
                    cbfi.setSearchValue(country);
                    cbfi.setSearchCriteria(countryMatchCondition);
                    setProps.put("COUNTRY",new CBNull( ));
                } else
                    setProps.put("COUNTRY",new CBNull( ));
            }
            if(population != null) {
                // Use the CBFieldInfo to specify the match condition
                if(populationMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
```

-continued

```
                    cbfi.setSearchValue(population);
                    cbfi.setSearchCriteria(populationMatchCondition);
                    setProps.put("POPULATION",cbfi);
                } else
                    setProps.put("POPULATION",population);
            }
            else if(populationNullIndicator) {
                // Use the CBFieldInfo to specify the match condition
                if(populationMatchCondition != -1) {
                    CBFieldInfo cbfi = new CBFieldInfo( );
                    cbfi.setSearchValue(population);
                    cbfi.setSearchCriteria(populationMatchCondition);
                    setProps.put("POPULATION",new CBNull( ));
                } else
                    setProps.put("POPULATION",new CBNull( ));
            }
            if(cbsqlwhere != null)
                setProps.put("CBSQLWHERE", cbsqlwhere);
            cqi.setFieldValues(setProps);
            return cqi;
        }
        // Instance Variables.
        protected String name = null;
        protected boolean nameNullIndicator = false;
        protected int nameMatchCondition = -1;
        protected String country = null;
        protected boolean countryNullIndicator = false;
        protected int countryMatchCondition = -1;
        protected Integer population = null;
        protected boolean populationNullIndicator = false;
        protected int populationMatchCondition = -1;
        String cbsqlwhere = null;
        CBQueryInfo cqi = new CBQueryInfo( ); // CBProp custom query
structure
        public Properties setProps = new Properties( ); // CBProp properties,
global for convenience!
        /**
         * Check to see if values are equal - this should be based on key
fields.
         * @param n_population the State object to compare to.
         */
        public boolean equals(Object t_Cityinfo)
        {
            if(! (t_Cityinfo instanceof Cityinfo)) return false;
            Cityinfo t_CompareCityinfo = (Cityinfo)t_Cityinfo;
            if(t_CompareCityinfo.getName( ) == null) {
                if(name != null) return false;
            }
            else
                if(!t_CompareCityinfo.getName( ).equals(name))
                    return false;
            return true;
        }
        /**
         * Get the CbSQL special Query By example value.
         * @return population Returns the qbe attribute value is current set
to.
        */ public String getCbsqlwhere( )
        {
            return cbsqlwhere;
        }
        /**
         * set the CbSQL special Query By example value.
         * @param population sets the qbe attribute value is current set to.
        */ public void setCbsqlwhere(String n_cbsqlwhere)
        {
            cbsqlwhere= n_cbsqlwhere;
        }
    }
```

Transparent Persistence with CocoBase

CocoBase accomplishes transparent persistence with java object models without using bytecode manipulation, proprietary interfaces or class hierarchy intrusion. This means that no special classes or interfaces are needed in the Object

Model in order to do persistence with CocoBase. The only requirement is that they must have a default constructor with no arguments.

There are 3 basic Runtime components that are involved in the transparent persistence of objects:

- The CocoBase Runtime O/R mapping class that wrappers the JDBC driver and issues queries and does the actual persistence calls. This is a class such as thought.CocoBase.CocoPowder or thought.CocoBase.CocoPowderPlugin20 (for jdbc 2.0 connections).
- The thought.CocoBase.Transaction object that can track changes of instances, and acts as a change 'buffer'. If a Transaction object is used, then it only calls a CocoBase Runtime driver—O/R mapping runtime class—when the txn.commit( ) is called.
- The thought.CocoBase.navapi.Navigator object that can track and detect changes in relationships based on Link definitions. Note that unlike CocoBase Maps, Link models are not kept in the CocoBase repository. Once a link definition model is created it is saved in the demos/resources directory (by default—although this can be overridden). As long as the model is in the classpath either directly or in a subdirectory called 'resources' it will find the model properties file and retrieve the navigation information.

The Navigator class can function in conjunction with a Transaction object or it can function standalone. While complex object graphs can be transparently managed directly by the Navigator and without the Transaction object, the use of the Transaction object is generally preferred because of its buffering and update optimizations which only persist those attributes that have changed.

Creating Applications with CocoBase Transparent Persistence

After classes have been generated and compiled, such as with the above application, CocoBase runtime classes can be used to persist instances of these classes. As shown in numberous other documents in the art, the CocoBase Navigator can easily develop a navigation model for the application. This model can then be used to persist the objects.

The Example below is not directed to persistence of the Navigation model for the above Cityinfo application, but the concepts can be easily applied to the Cityinfo application to provide transparent persistence.

First, open a CocoBase connection as follows:

```
CocoDriverInterface myBase = CocaDriver.getCocoDriver(
    "thought.CocoBase.CocoPowder",
    "org.hsql.jdbcDriver",
    "jdbc:HypersonicSQL:hsql://localhost;
    cocoprop=cocofactory=CocoProxyFactory",
    "sa", ""
)
if(myBase.connect( ) == -1) {
    System.out.println("Failed connect!");
    System.exit(1);
}
. . .
```

Now create a CocoBase transaction object to manage any objects that are retrieved. Notice how the transaction object is configured through parameters and property settings.

```
thought.CocoBase.Transaction cocoTxn =
    new thought.CocoBase.Transaction(myBase, false);
Properties cocoTransProps = new Properties( );
cocoTransProps.put("preserveCommit","true");
cocoTransProps.put("commitconnection","true");
cocoTransProps.put("throwExceptions","true");
cocoTransProps.put("updateOnlyChangedColumns","true");
cocoTxn.setProperties(cocoTransProps);
// Begin a new transaction.
cocoTxn.begin( );
. . .
```

Then open a CocoBase Navigator object with the Navigation model to be used and also we register the Transaction object with the Navigation model:

```
// Instantiate a Navigator with the Link model created from
// the UML/XMI document
thought.CocoBase.navapi.Navigator navigator =
    new thought.CocoBase.navapi.Navigator (myBase, "company");
// Assign the current transaction to the Navigator
navigator.setTransaction(cocoTxn);
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
// Setup our object to query
Department dept = new Department( );
department.setName("SALES");
// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = myBase.selectAll(
            new thought.CocoBase.CocoProxyM(dept), "Department");
```

The CocoProxyM is optional, the code example below shows the same call without its usage.

```
// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = myBase.selectAll(dept, "Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the loadAllLinks method which does the automatic navigation for that object:

```
for(int i=0; i< deptVector.size( ); i++) {
    Department d = (Department)deptVector.elementAt(i);
    // Because the cascadeLoad flag is set to true in the direction
    // Department->Employees, the employees link will load
    automatically
    d = navigator.loadAllLinks(d,"Department");
    . . .
    Vector emps = d.getEmployees( );
    for (int j=0; j<emps.size( ); j++) {
        // raise salaries by 20%
        emp.setSalary(emp.getSalary( )*1.2);
        . . .
    }
```

-continued

```
    . . .
    // Once changes are made to an object graph those changes can be
    // synchronized using the updateAllLinks method such as:
    navigator.updateAllLinks(d,"Department",true);
}
```

You can then commit the buffered changes with:

cocoTxn.commit( );

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-J2EE applications the application intrusion is incredibly small, requiring a single method call for each root node.

The Navigator supports one-to-one, one-to-many and many-to-many relationships with cycle detection. It also detects this locally (i.e. in the client application) or by reconciling serialized or copied objects without any kind object model or bytecode intrusion. This is truly transparent persistence that is architected and designed for the VM oriented Java language. There are also more advanced applications included in the demos/pguide/navapi subdirectory that demonstrate one-to-one, one-to-many and many-to-many relationships as well as an EJB using the Navigator system to manage a graph of java objects.

The Example below is not directed to persistence of the Navigation model for the above Cityinfo application, but the concepts can be easily applied to the Cityinfo application to provide transparent persistence.

First, open a CocoBase CBFacade Transparent Persistence connection as follows:

```
// Set up the connection class - can be Local or Remote with EJB Server
CBFacade cbf =
CBFacade.create("thought.CocoBase.CBFacadeLocal");
Properties props = new Properties( );
props.put("cocosource.name",args[0]);
props.put("cocosource.jdbcdriver",args[1]);
props.put("cocosource.url",args[2]);
props.put("cocosource.user",args[3]);
props.put("cocosource.password",args[4]);
props.put("cocosource navmodel","CompanyAppLinks");
cbf.connect(props);
. . .
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
    // Setup our object to query
    Department dept = new Department( );
    department.setName("SALES");
    // This will read & bind all 'Department' objects to the transaction.
    Vector deptVector = cbf.loadAll(dept,"Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the loadAll method which does the automatic navigation for that object:

```
for(int i=0; i< deptVector.size( ); i++) {
    Department d = (Department)deptVector.elementAt(i);
    // Because the cascadeLoad flag is set to true in the direction
    // Department->Employees, the employees link will load
    automatically
    d = cbf.loadAll(d,"Department");
    . . .
    Vector emps = d.getEmployees( );
    for (int j=0; j<emps.size( ); j++) {
        // raise salaries by 20%
        emp.setSalary(emp.getSalary( )*1.2);
        . . .
    }
    . . .
    // Once changes are made to an object graph those changes can be
    // synchronized using the save method such as:
    cbf.save(d,"Department");
}
```

You can then commit the buffered changes with:

cbf.commit( );

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-J2EE applications the application intrusion is incredibly small, requiring a single method call for each root node.

Both the above architecture and two implementations presented herein are uniquely suited to work in every application from the tiny local app to the enterprise J2EE and to do so with superior performance and manageability.

I claim:

1. A system for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising:

a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, wherein the set of definitions is stored in a repository;

b) a set of definitions for the relationships between objects for an object language application, wherein the set of definitions is part of an object application navigation model;

c) a list of objects, or a set of objects that are to be persisted, wherein the list of objects or set of objects is part of an object application navigation model;

d) an object or set of objects as part of a computer implemented method that contains the computer logic capable of persisting an indicated object or set of objects;

e) an input method to provide the computer logic of d) with data corresponding to the location of information related to a), b) and c), and

f) at least one data source to in which persisted data may be stored.

2. A system according to claim **1**, that does not require any modifications to an object model or the inclusion of any

persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model.

3. A system according to claim **2**, that provides persistence for a complex data object graph model on a distributed network environment.

4. A system according to claim **3**, wherein the displayable presentation page is a JSP that is associated with a first Java object referenced within the JSP source code, which is a JSP processor class object, that is in turn associated with a second Java object referenced by the first Java Object with embedded business logic that reflects the data of the first business object and can be tailored to provide additional business logic for forward and reverse conversion of data types acceptable for the JSP into data types to be stored in a data source.

5. A system according to claim **2**, that provides persistence for a complex data object graph model on a distributed network environment, further comprising a generic ejb stateful session bean.

6. A system according to claim **1**, further providing a system for displaying, updating or creating the data or structure of one or more data objects, and optionally its links to other objects of an object model, by utilizing a displayable presentation page format wherein the displayable presentation page source code contains embedded object programming logic, and persisting one or more of a member selected from the group consisting of an object, the data of an object, links of an object, or an object model by having the object programming logic that is embedded in the displayable presentation page directly or indirectly communicate with at least one object having at least one computer implemented method that includes imbedded computer logic for getting, setting, resetting and loading of data, and wherein the at least one object also includes computer implemented logic for communicating directly or indirectly with a persistence manager API or persistence library to persist one or more members selected from the group consisting of an object, the data of an object, a link to another object in an object model, and an object model.

7. A system according to claim **1**, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, further comprising a generic ejb stateful session bean.

8. A system according to claim **1**, that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model, further comprising a generic ejb stateful session bean.

9. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method to create or maintain transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising:

i) logic for a computer implemented method for creating a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, wherein the logic for the computer implemented method is capable of causing the storage of the set of definitions in a repository;

ii) logic for a computer implemented method for creating and accessing a set of definitions for the relationships between objects for an object language application, wherein the logic for the computer implemented method is capable of causing storage of the set of definitions in a navigation model;

iii) logic for a computer implemented method providing an input means for a member selected from an object name, a list of objects, or a set of objects that are to be persisted, wherein the logic for a computer implemented method is capable of causing the storage of the input as part of a navigation model that can be accessed by ii);

iv) logic for a computer implemented method containing the logic and methods to persist an object or set of objects based upon access to (a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, (b) a set of definitions for the relationships between objects for an object language application, and (c) an object name, a list of objects, or a set of objects that are to be persisted, and

v) logic for a computer implemented method that includes an input method which is capable of and has a method for informing iv) of the location of information relating to iv)(a), iv)(b) and iv)(c).

10. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method according to claim **9**, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising software code for generating, accessing, or maintaining a generic ejb stateful session bean object.

11. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method according to claim **9**, that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model, further comprising a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method of generating, accessing, or maintaining a generic ejb stateful session bean object.

12. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **9**, that provides persistence for a CDOG model on a distributed network environment.

13. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **9**, that includes a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method to

automatically generate the logic for at least one of the routines from a relational data source schema, or to generate the logic from an object to relational mapping resource, said compiled or programming language software logic object code routine or set of object code routines that are stored in a permanent medium or in a computer system memory for a computer implemented method comprising:

a). at least one presentation page having embedded within its source code object language programming logic that references attributes of a data object and also references at least one associated programming object for executing commands associated with navigating the presentation page, wherein the presentation page has an ability to display the data of an object in an object model, and to optionally display information regarding at least one link of that object to another object in an object model or to provide a labeled link button for navigating to the presentation page of a linked object in an object model, or to provide both a link and a labeled link button, and

b). at least one associated corresponding programming object containing programming logic for communicating with the presentation page and executing commands associated with such navigation, and further containing logic for directly or indirectly communicating with a persistence manager API, or persistence library, that as the presentation page is being navigated can persist the object, its data or links to other objects in an object model, or both its data and links to other objects in an object model, and

c). optionally comprising a set of related presentation pages that can be navigated and their associated programming objects as described in (a) and (b) above, which collectively represent the linked objects of a specified object model, and

d). optionally comprising an a computer implemented method with the ability to query against the object graph using a object query facility or syntax.

14. A method of claim **13**, comprising a set of related presentation pages that can be navigated and their associated programming objects as described in (a) and (b), which collectively represent the linked objects of a specified object model.

15. A method according to claim **14**, wherein the object query facility is a computer implemented method and associated logic that utilizes EJB-QL query syntax or similar syntax.

16. A method according to claim **13**, wherein the object query facility is a computer implemented method and associated logic that utilizes EJB-QL query syntax or similar syntax.

17. A method for visually navigating an object model while dynamically creating, updating and persisting such an object model to a data source by utilizing the compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method of claim **13**, whereby a navigation model may be navigated in at least one direction and any changes to one or more of members selected from the group consisting of the data of an object, an object, a link between an object, or an object model can be dynamically persisted to a data source.

* * * * *

EXHIBIT F

US007149730B2

(12) **United States Patent**
Mullins et al.

(10) **Patent No.: US 7,149,730 B2**
(45) **Date of Patent: *Dec. 12, 2006**

(54) **DYNAMIC CLASS INHERITANCE AND DISTRIBUTED CACHING WITH OBJECT RELATIONAL MAPPING AND CARTESIAN MODEL SUPPORT IN A DATABASE MANIPULATION AND MAPPING SYSTEM**

(76) Inventors: **Ward Mullins**, 2222 Leavenworth St. Apt. 304, San Francisco, CA (US) 94133; **Alexandre Martins**, Rua Almirante Lamego 703 Apt. 102, Centro, Floianopolis SC (BR) 88015-600

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 306 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/430,004**

(22) Filed: **May 5, 2003**

(65) **Prior Publication Data**

US 2003/0208505 A1 Nov. 6, 2003

Related U.S. Application Data

(60) Provisional application No. 60/377,879, filed on May 3, 2002.

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/2**; 707/3

(58) **Field of Classification Search** 707/1, 707/3, 4, 5, 10, 100, 102, 103, 104.1, 2; 345/736; 711/141; 709/201, 203, 225, 251; 713/201; 717/108, 114, 162, 165
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,754,670 B1 * 6/2004 Lindsay et al. 707/100

OTHER PUBLICATIONS

BEA WebLogic Server 5.1 Documentation, copyright 2000, pp. 1-63.*

* cited by examiner

Primary Examiner—Apu M. Mofiz

(57) **ABSTRACT**

The present invention provides a system and method for dynamic object-driven database manipulation and mapping system which relates in general to correlating or translating one type of database to another type of database or to an object programming application. Correlating or translating involves relational to object translation, object to object translation, relational to relational, or a combination of the above. Thus, the present invention is directed to dynamic mapping of databases to selected objects. Also provided are systems and methods that optionally include caching components, security features, data migration facilities, and components for reading, writing, interpreting and manipulating XML and XMI data files.

16 Claims, 7 Drawing Sheets

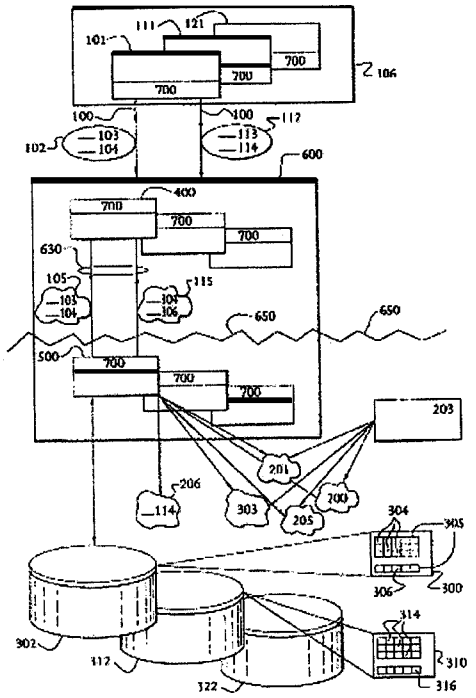


Figure 1

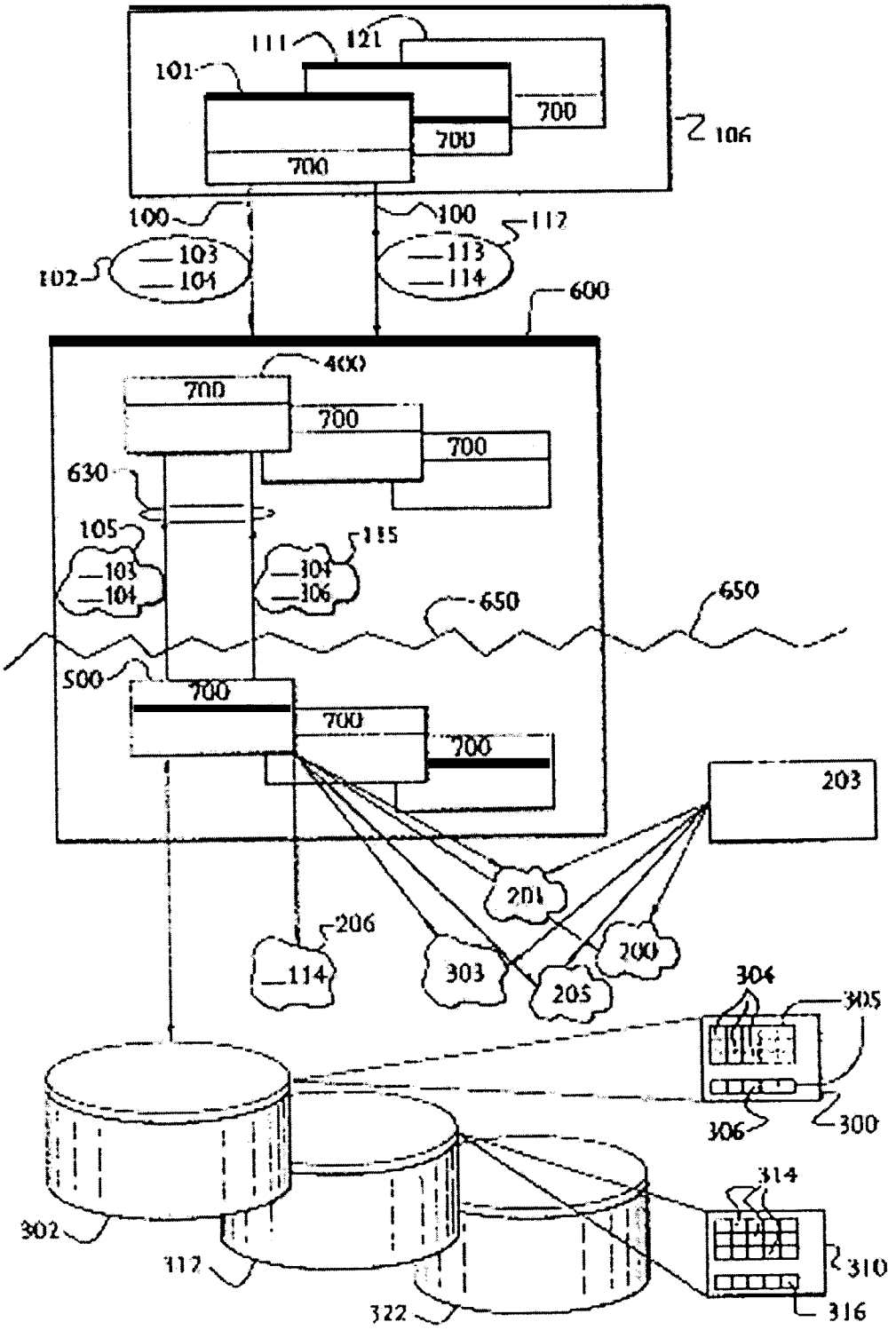

111
121
101
700
700
700
106
100
100
117
102
103
104
113
114
600
400
700
700
700
630
105
103
104
115
304
306
650
650
500
700
700
700
203
201
206
114
303
200
205
304
305
306
300
314
302
312
310
322
316

Figure 2

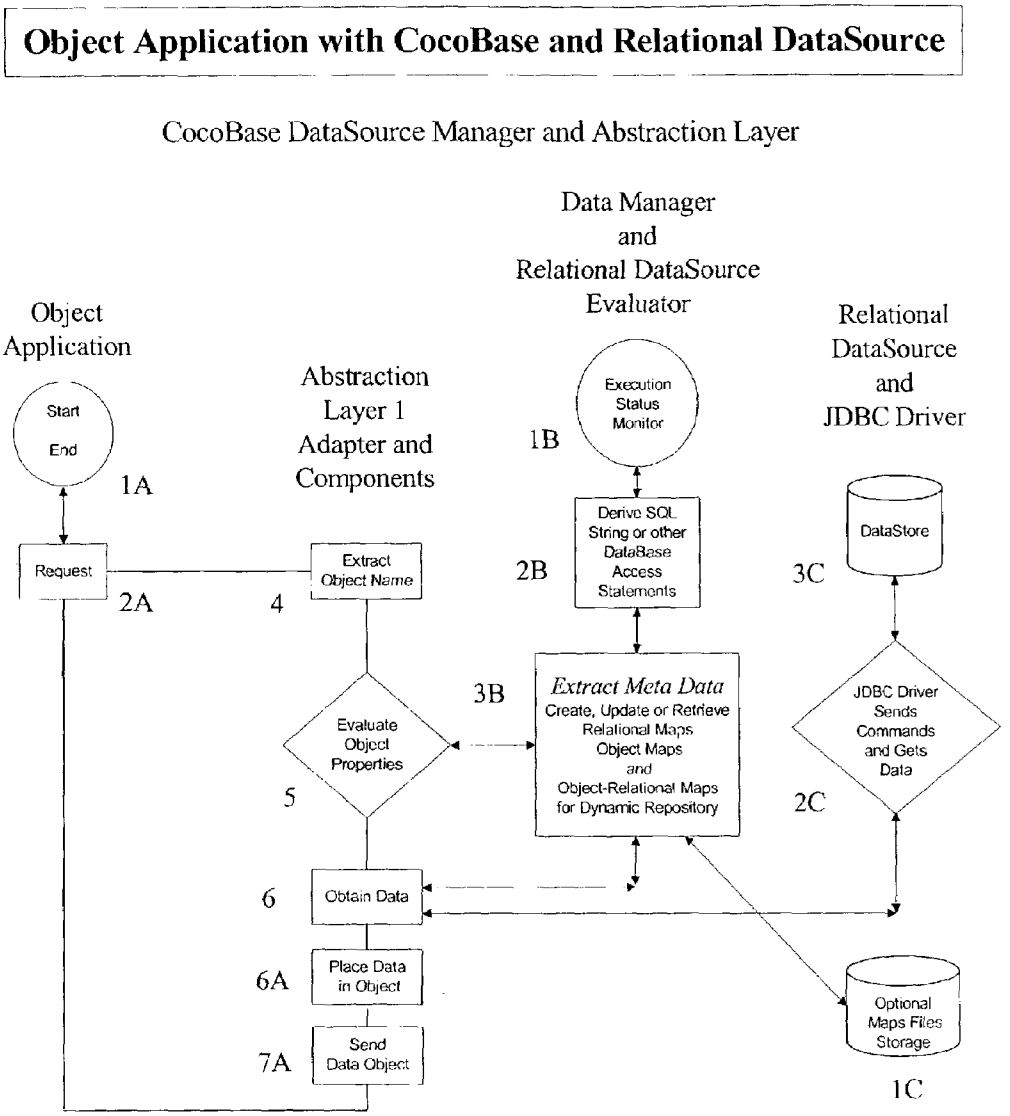

Object Application with CocoBase and Relational DataSource
CocoBase DataSource Manager and Abstraction Layer
Object Application
Start
End
1A
Request
2A
Abstraction Layer 1 Adapter and Components
Extract Object Name
4
Evaluate Object Properties
5
Obtain Data
6
Place Data in Object
6A
Send Data Object
7A
Data Manager and Relational DataSource Evaluator
Execution Status Monitor
1B
Derive SQL String or other DataBase Access Statements
2B
3B
Extract Meta Data
Create, Update or Retrieve Relational Maps Object Maps and Object-Relational Maps for Dynamic Repository
Relational DataSource and JDBC Driver
DataStore
3C
JDBC Driver Sends Commands and Gets Data
2C
Optional Maps Files Storage
1C

Figure 3

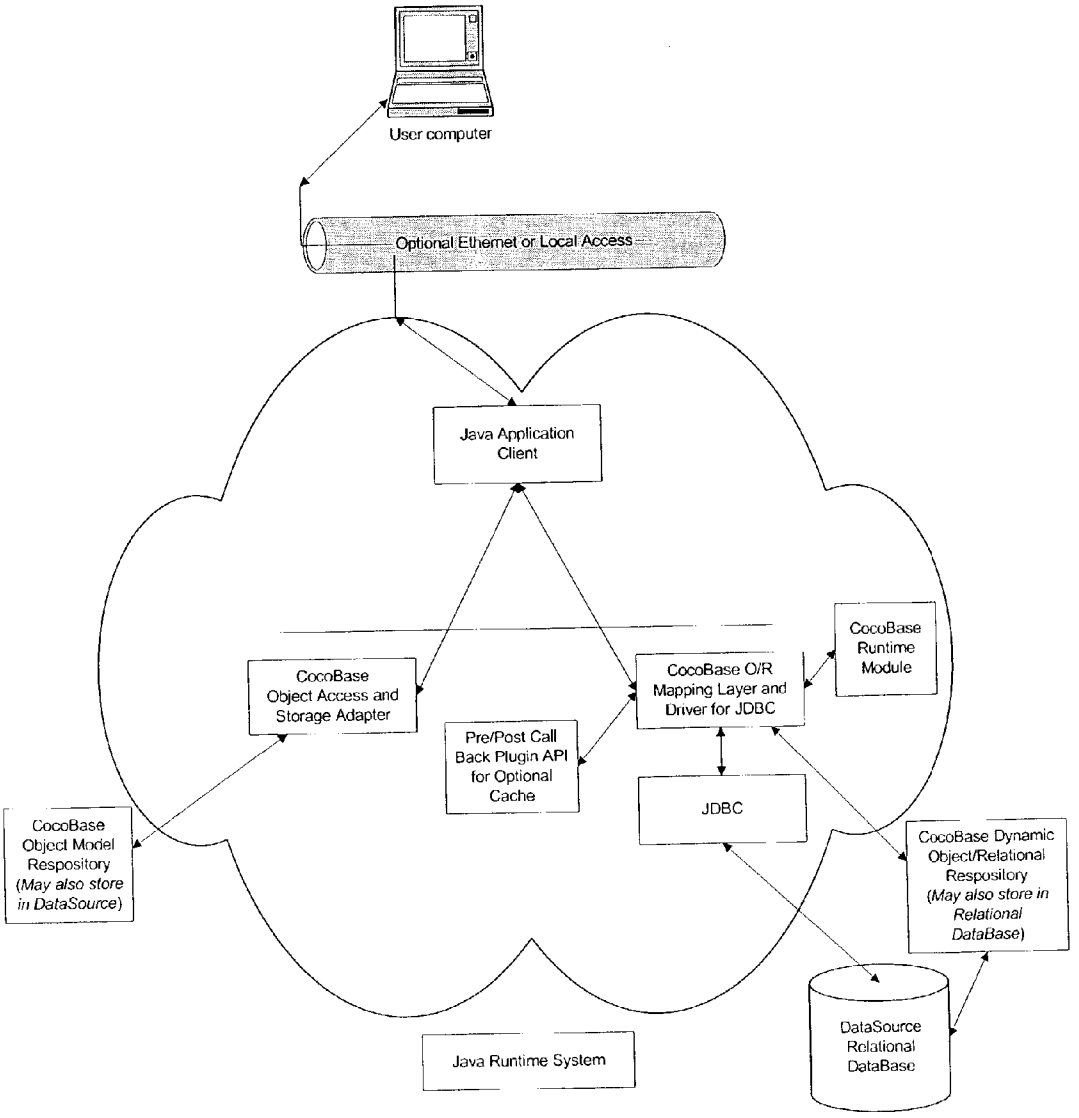

User computer
Optional Ethernet or Local Access
Java Application Client
CocoBase Object Access and Storage Adapter
CocoBase O/R Mapping Layer and Driver for JDBC
CocoBase Runtime Module
Pre/Post Call Back Plugin API for Optional Cache
JDBC
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
DataSource Relational DataBase
Java Runtime System

Figure 4

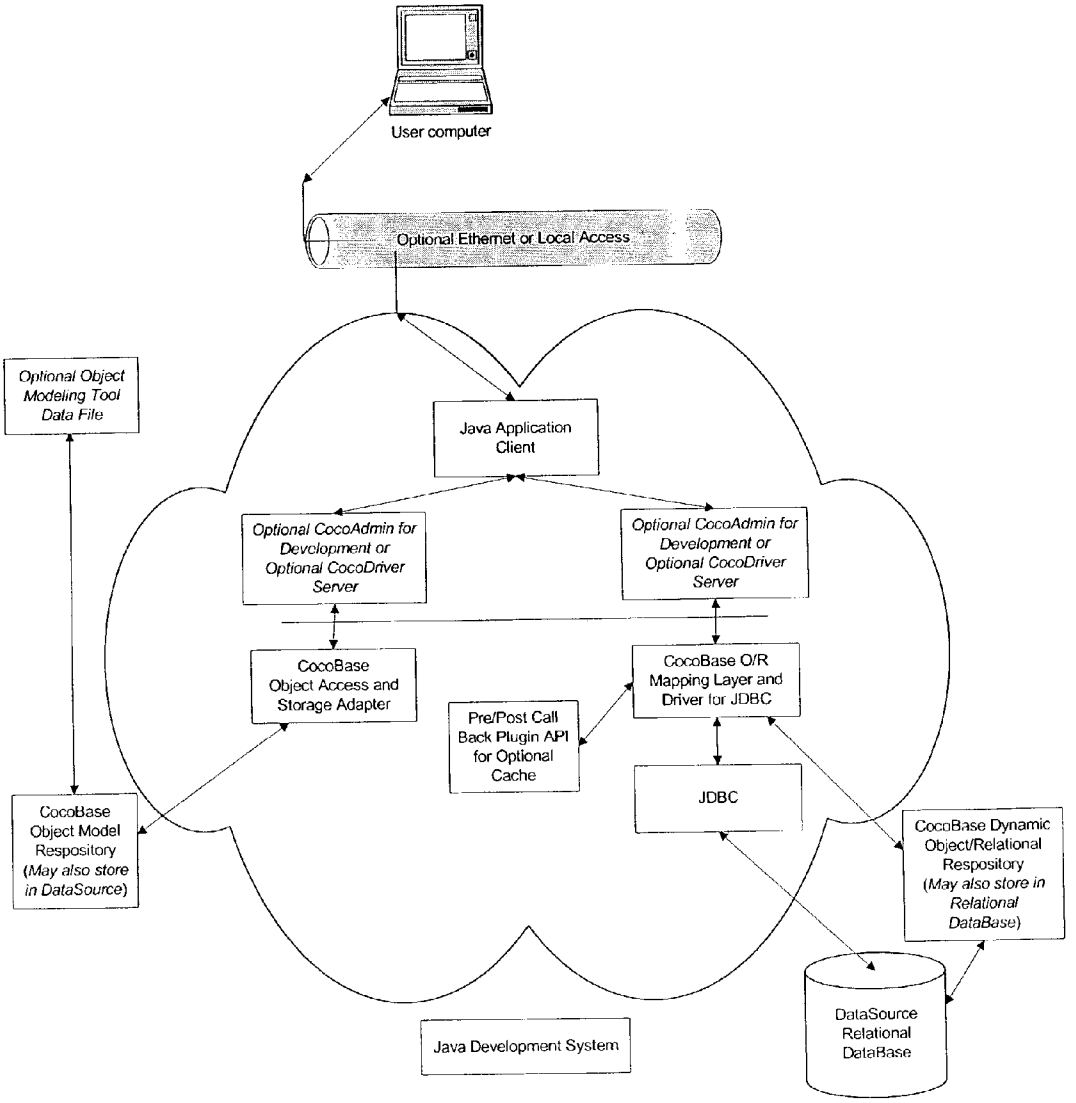

User computer
Optional Ethernet or Local Access
Optional Object Modeling Tool Data File
Java Application Client
Optional CocoAdmin for Development or Optional CocoDriver Server
Optional CocoAdmin for Development or Optional CocoDriver Server
CocoBase Object Access and Storage Adapter
CocoBase O/R Mapping Layer and Driver for JDBC
Pre/Post Call Back Plugin API for Optional Cache
JDBC
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
DataSource Relational DataBase
Java Development System

Figure 5

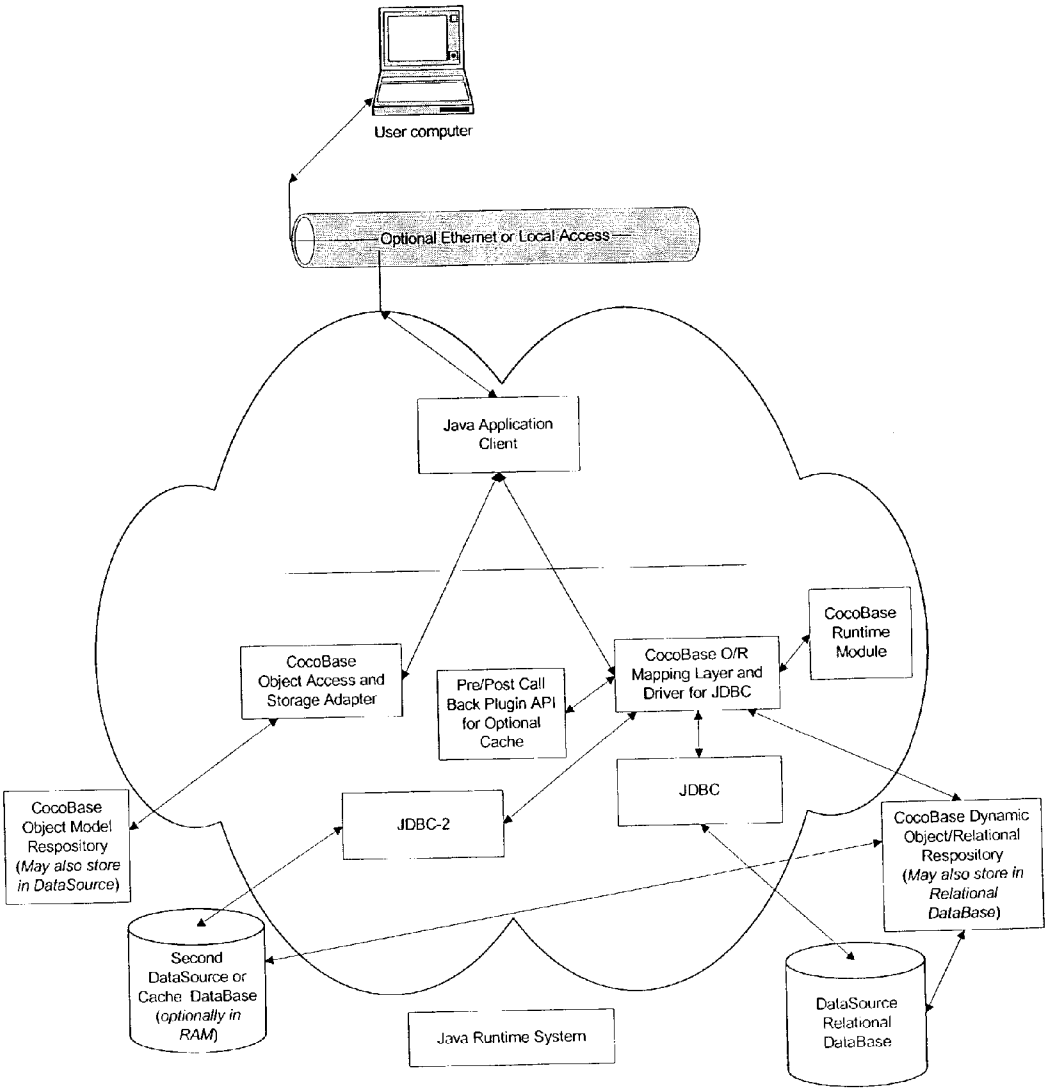

User computer
Optional Ethernet or Local Access
Java Application Client
CocoBase Object Access and Storage Adapter
Pre/Post Call Back Plugin API for Optional Cache
CocoBase O/R Mapping Layer and Driver for JDBC
CocoBase Runtime Module
JDBC
JDBC-2
CocoBase Object Model Respository (May also store in DataSource)
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
Second DataSource or Cache DataBase (optionally in RAM)
Java Runtime System
DataSource Relational DataBase

Figure 6

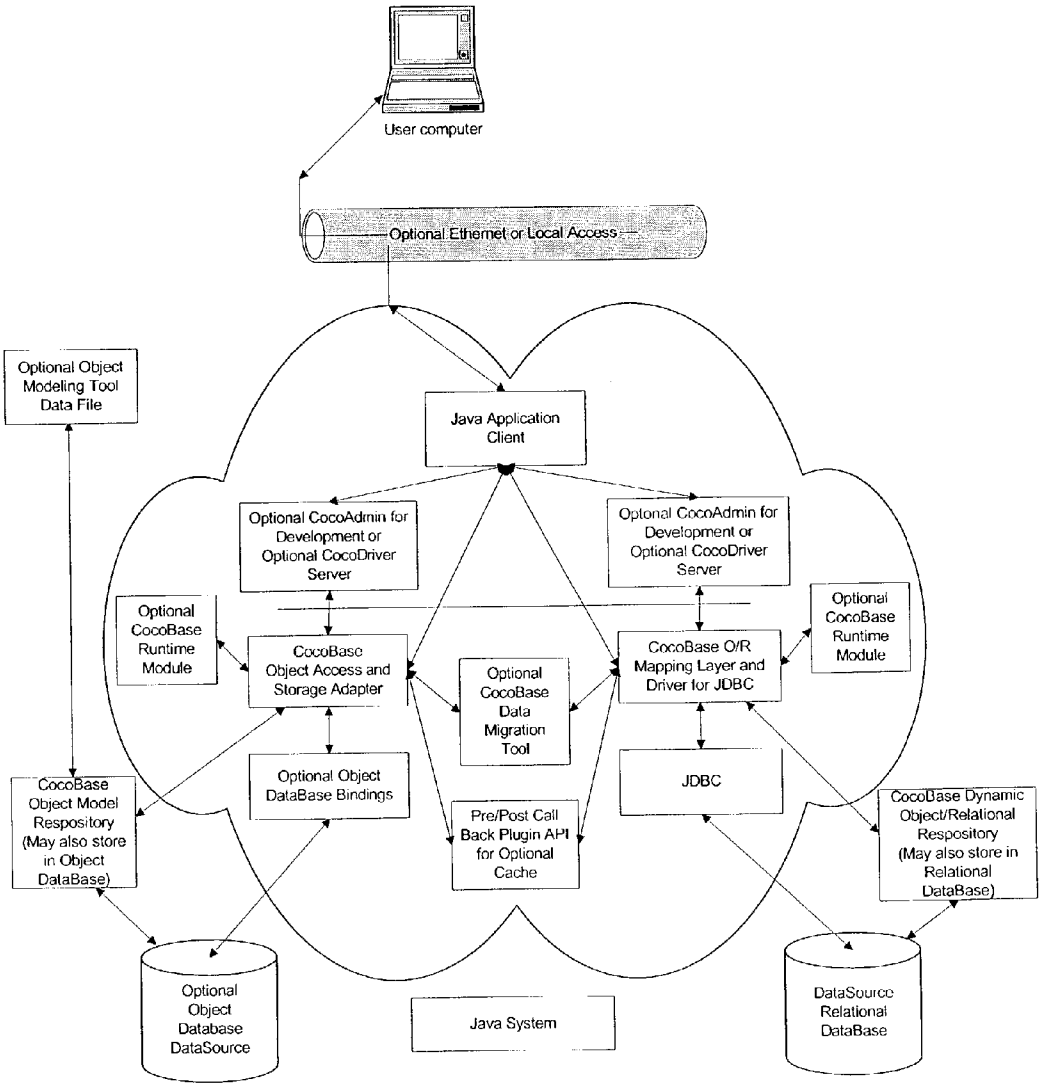
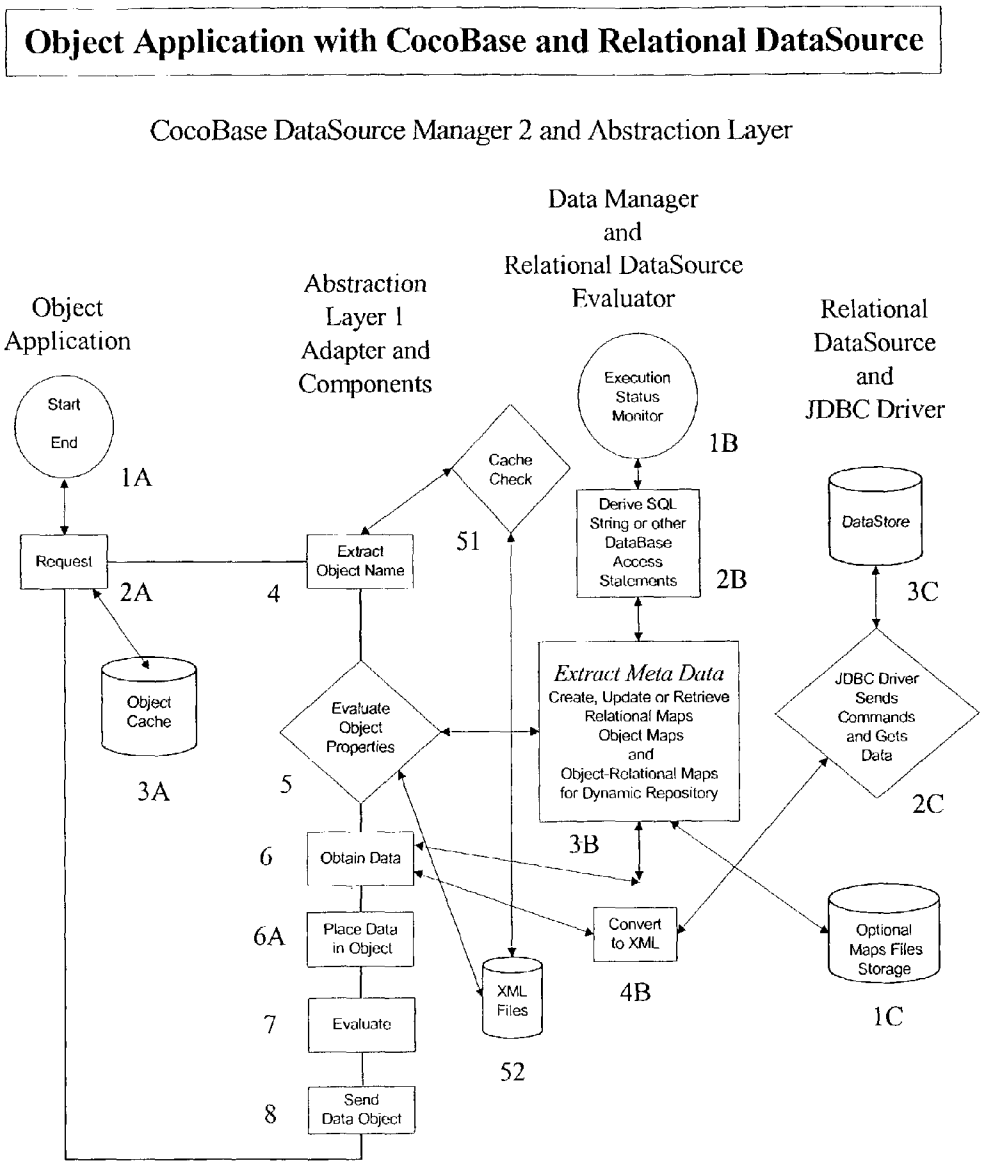

User computer
Optional Ethernet or Local Access
Optional Object Modeling Tool Data File
Java Application Client
Optional CocoAdmin for Development or Optional CocoDriver Server
Optonal CocoAdmin for Development or Optional CocoDriver Server
Optional CocoBase Runtime Module
CocoBase Object Access and Storage Adapter
Optional CocoBase Data Migration Tool
CocoBase O/R Mapping Layer and Driver for JDBC
Optional CocoBase Runtime Module
Optional Object DataBase Bindings
JDBC
CocoBase Object Model Respository (May also store in Object DataBase)
Pre/Post Call Back Plugin API for Optional Cache
CocoBase Dynamic Object/Relational Respository (May also store in Relational DataBase)
Optional Object Database DataSource
Java System
DataSource Relational DataBase

Figure 7


Object Application with CocoBase and Relational DataSource
CocoBase DataSource Manager 2 and Abstraction Layer
Object Application
Abstraction Layer 1 Adapter and Components
Data Manager and Relational DataSource Evaluator
Relational DataSource and JDBC Driver
Start
End
1A
Request
2A
Object Cache
3A
Extract Object Name
4
Cache Check
51
Execution Status Monitor
1B
Derive SQL String or other DataBase Access Statements
2B
DataStore
3C
Extract Meta Data
Create, Update or Retrieve Relational Maps Object Maps and Object-Relational Maps for Dynamic Repository
3B
JDBC Driver Sends Commands and Gets Data
2C
Evaluate Object Properties
5
6
Obtain Data
6A
Place Data in Object
7
Evaluate
8
Send Data Object
XML Files
52
Convert to XML
4B
Optional Maps Files Storage
1C

DYNAMIC CLASS INHERITANCE AND DISTRIBUTED CACHING WITH OBJECT RELATIONAL MAPPING AND CARTESIAN MODEL SUPPORT IN A DATABASE MANIPULATION AND MAPPING SYSTEM

This new application is for an Original (nonprovisional) filing, based on U.S. Provisional Application Ser. No. 60/377,879, filed May 3, 2002

TECHNICAL FIELD

The present invention relates in general to correlating or translating one type of database to another type of database or to an object programming application. Correlating or translating involves relational to object translation, object to object translation, relational to relational, or a combination of the above. Thus, the present invention is directed to dynamic mapping of databases to selected objects in a system that supports dynamic class inheritance and distributed caching with either or both of Object to Relational and Relational to Object mapping, and including Cartesian model mapping support and class inheritance model mapping support in a database manipulation and mapping system.

BACKGROUND ART

Computer databases or similar constructions (hereinafter referred to as data stores) are powerful tools for storage, organization, retrieval and other handling of various types of information. However, there are different database models, or formats, for data access that are incompatible with each other, and may also be incompatible with, or remarkably different from, an object programming application. In this respect, complex relationships between objects present in the object programming application may be entirely absent in a relational or object database being accessed or updated. Nonetheless, a plurality of these database types have achieved a high level of popularity and proliferation. The two most common database models are object and relational, and object programming models are frequently used for internet applications involving database accessing.

As an example of common object programming applications, the popular Java language with its server-side component, Enterprise Java Beans, is being widely used to support object model programming applications that access and use data from databases that have a JDBC driver. Thus, it is desirable to be able to adapt and use many of the traditional relational database tools, utilities, systems and corporate procedures with the newer object format of many more recent web-based applications. Since many users may be accessing a single application, there is a need for speeding up database accesses and for synchronizing object programming application transactions with database accesses and updates.

Systems for accessing data storage based upon objects have been used for many years to accommodate object-directed software applications. The most common conventional approach to accomplish access of data stores involves writing and imbedding custom access codes within an object application needing access to the data. This approach is generally limited to having custom codes capable of accessing only a single relational table within a relational database or similar construction, referred to as a data store. When an object model has been adapted specifically for accessing a single data store using the JDBC driver for that data store,

moving data to a second database or accessing multiple databases can become very problematic and painful for a programming engineer who is assigned the responsibility for such a task.

The aforementioned conventional approach to bridging between object and relational models (or between object and multiple relational models) has frequently required "brute-force" hand-recoding of relational database access code. This approach can be very time-consuming and is prone to errors. This approach can also generate inefficient objects that require more processing time, storage space, and transfer time. Further, this approach does not scale well because each instance of relational access code must be converted manually, in turn, even if similar objects have already been applied for use with the converted relational data base.

As mentioned above, more efficient approaches sometimes use tools to automate the conversion process. However, the tools provided to date typically are not versatile enough to handle many types of conversion without manual coding or explicit defining of parameters by a human operator. Because of this requirement for operator participation in the translation, the conversion using today's tools still does not scale well and does not allow for fully automated and flexible conversion of data access between various data base models.

Further, this approach to conversions does not provide for communicating the conversions in an efficient manner as, for example, to other entities who may need the same type of conversion on similar data, or who may require the conversion to access the data in another way. In a distributed environment that may involve multiple users accessing multiple databases by multiple object applications, this can get exceeding complex. Another drawback of conventional systems and techniques (as understood in the conventional art), is that dynamic mapping of objects to multiple types of databases is virtually impossible, because the tailored hand code must be updated and recompiled. This may require that an application be stopped and restarted in order to implement mapping changes, i.e., the mapping changes are not dynamic.

Significant drawbacks are associated with using a relational database model in conjunction with an object programming model, but the relational database model is still the most widely accepted traditional format for reliable and quick access to data while using complex data search queries. This model allows the user to store data records and to access the data via a query language that uses relational expressions such as AND, OR, NOT, etc. Over the years, the vast majority of production databases have been relational databases, each having its own relational arrangement, and accessible by a particular query language. While convenient for set-up and arranging data, access is had only by those having substantial knowledge of the various relationships between pieces of data in the database.

JDBC drivers have been designed to formulate database specific SQL queries or statements from standard SQL strings passed to them by an object programming application. Ordinarily, such JDBC drivers are specific to a database, and separate statements and separate JDBC drivers must be utilized to access different databases. This is complicated by the fact that many object programming applications either do not permit simultaneous attachment to multiple databases or poorly coordinate transactions that impact multiple databases. Accordingly, relational databases can sometimes be awkward to access, and transferring data from one database to another database can be slow and painful,

unless a specific transfer program has been specifically written or tailored to effect that specific transfer of data.

The object database model is a newer model that has rapidly increased in usage due to recent trends such as the world-wide-web and application service provider (ASP) architectures. Such object databases have sometimes been used as the primary source for accessing data, for example, Jasmine and ObjectStore are available from Computer Associates Intl., Inc. and Object Design Inc. (ODI), respectively. However, accessing data in such permanently stored object database has raised serious performance issues for larger applications when complex querying is required (as compared to relational databases). Complex querying is usually faster using relational databases since query engines may be more highly developed for relational database accesses.

Previously, object applications have tended to only use simple memory buffers with relational and object databases. Difficulties existed with using either an object database or a relational database to serve as a transient memory resident database "cache" (secondary database) to try to speed up database accesses. No commercially viable solutions have existed with acceptable performance.

There is a need for an object programming application to successfully access a secondary transient memory resident database, as either a relational database or an object database. In such a case, a dynamic database mapping tool would be necessary to provide the object programming application with a map or maps to both the primary and secondary databases in order to successfully implement a dynamic transient memory resident database. It would be particularly helpful in optimizing the performance of an object programming application if there could be provide a memory resident object database (or a resident portion thereof) as a cache for data retrieval. In such a case it might be possible to directly access (and perhaps re-use) data objects stored in the secondary transient resident memory database, including their metadata, instead of generating data objects from accessed data.

If it were possible to successfully provide a commercially viable secondary memory resident transient database, data being used by the object programming application would only need to be persisted back to one or more of a permanent storage relational database (primary database) when data has been created, inserted or updated. Ideally, the application would likewise update the memory resident object database (secondary database) when the primary database is updated. However, as mentioned such simultaneous access of multiple databases can give rise to transactional coordination problems, or to data synchronization problems.

Accordingly, intensely needed by many modem software application programs are systems for more efficient access and manipulation of data stores, systems having the flexibility and dynamic capability to attach data from a database to maps as objects and having the ability to map one or more databases to various objects in real time. A strong need exists for such systems that also permit a user to cleanly, transparently and synchronically transfer data between multiple data sources, while maintaining the ability for an object programming application to access or use such data in the system. The need exists for an improved database access buffering system, for object programming applications, having at least one transparent secondary memory resident database and a primary data source that can be simultaneously utilized in a synchronized and transaction coordinated manner.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the present invention to overcome the drawbacks of the conventional art and achieve flexibility and dynamic operation in the accessing of databases that has not yet been achieved on a consistent basis while providing dynamic class or data persistence and/or dynamic object model navigation with distributed caching of a data source that uses a object to relational or a relational to object mapping respository.

It is another object of the present invention to provide a system for dynamically mapping data to objects for software applications, particularly at a system for mapping to an object for a software application during the run time of that application, in a system that provides dynamic persistence of data and navigation of an object model,.

It is an additional object of the present invention to provide easy translations between databases and applications having a variety of different formats or data store models, and to provide for class inheritance and Cartesian model support in a mapping system, preferably and particularly in a system that provides distributed caching of a datasource and/or of an object programming navigation model that is mapped to a datasource.

It yet another object of the present invention to provide a mapping system wherein multiple objects can be used for a particular map, and preferably providing parent-child two way navigation and inheritance as well as Cartesian model support in a mapping system.

It is still a further object of the present invention to provide a mapping system wherein multiple maps can be developed for a single object, for example in a parent-child inheritance situation where the object is populated by data present in both the parent class and child class maps, or the object is dynamically populated from multiple maps in a manner dictated by a Cartesian model.

It is again another object of the present invention to provide such a mapping system wherein different data attached to an object can be selected by means of different maps.

Another object of the present invention is to provide such a mapping system wherein object programming applications are tailored to delegate accessing a database and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanisms in the application code. It is a preferred object of the invention to provide such a system with a concurrent parameter setting mechanism where the runtime library repository can be set to access a particular database and to generate database specific database calls and SQL strings.

In one preferred object of the invention, in such a mapping system, is provided a software program or software module, such as an API, that can be optionally utilized when another software program or module is generating application programming code from database maps to provide a programming application that will delegate to a runtime library repository the functions of accessing a database and the generation of SQL strings that are specific to a database or JDBC driver.

In another preferred object of the invention, the runtime library repository can be modified and tailored to optimize database access calls and SQL strings for a particular database. Preferably, such repository supports both Cartesian models and Inheritance models for an object to relation, object to object or relational to object mapping system.

It is again another object of the present invention to provide a mapping system wherein data changes related to a particular object can be promulgated with global changes for that object, if desired. Particularly ideal is to provide for such global changes to a distributed cache or a distributed object model that is being used or navigated.

It is still a further object of the present invention to provide a mapping system when a data map for an object can be easily edited without extensive knowledge of the relational database as a source of the data.

It is again another object of the present invention to provide a mapping system wherein the metadata describing a map of a datastore can be dynamically evaluated or changed using a navigation object or set of navigation objects. Distributed caching of an object model to handle the existence of multiple object navigators is an ideal object of this invention.

It is again another object of the present invention to provide a mapping system, wherein data can be accessed more quickly than is possible with a conventional data store accessing arrangements.

It is again a further object of the present invention to provide a mapping system in which frequently-used data can be more easily accessed than other types of data.

It is yet a further object of the present invention to provide a mapping system in which a wide variety of different data languages can be easily used.

It is still a further object of the present invention to provide a mapping system wherein virtually any type of datastore architecture can be translated so as to be useful by an object software application, or other types of software applications.

It is again a further object of the present invention to provide a mapping system wherein datastore to datastore mapping is easily facilitated.

It is still another object of the present invention to provide a fully synchronized caching system that utilizes the transaction coordination facilities of a server such as a J2EE application server (for example, Weblogic 6.1), such system comprising a local or distributed computer system having a first data source referred to as the primary data source, a second data source referred to as the cache data source and the cache data source is associated with an object to relational mapping layer to provide a data source cache for object applications, and a server having a transaction coordinator with the ability to register multiple data sources, wherein:

(a) both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and

(b) the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and

(c) registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

In a particularly preferred object of the invention the cache data source can be set up as a memory-resident entire database, a disk resident database, or both a memory resident database and disk resident which are synchronized. In one object, of the invention the memory resident database may be a portion of the disk resident database and the size of the memory for the memory resident database can be set by the user.

Another object of the present invention is to provide a database access repository, or a collection of repositories, comprising a runtime library repository having the ability to make database specific calls and to generate database specific SQL strings, and comprising other libraries, information and logic that permit mapping of information related to one or more of a member selected from the group consisting of relational databases, object databases and object programming schemas, and the like. The database access repository may contain one or more items selected from the group consisting of maps of database schemas, object definitions, other programming definitions or parameters, object metadata, database settings, complex relationship definitions. For example, metadata can be placed in a separate structure from the java object application and can be independently stored and accessed in a database access repository file collection, or in a separate file. In one preferred object of the invention, an object to relational repository can include a runtime library repository that can be optimized for a particular database and has the ability to access multiple database types, wherein the runtime library repository can be access by object programming applications that delegate database access and SQL string generation to the runtime library repository.

These and other objects and goals of the present invention are achieved by the present invention as described below, and depicted in FIG. **7**.

In some objects of the present invention, concepts are based in part upon concepts present in U.S. Pat. No. 5,857,197, (incorporated herein by reference) or that are reasonably inferable from reviewing that patent, in order for the present invention to provide an improved mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about Java objects including complex java objects that have relationships with other java objects) does not need to be stored in a java object. In a preferred aspect, it is an object of the present invention to provide a database access system that does permits a java object application to delegate database accesses to a runtime library repository.

In a preferred object of the present invention, a software programming module (or modules) can automatically generate object source code from at least one database schema map, at least one object programming application schema, or from a combination of at least one database schema map and at least one object programming application schema. The code generated for the application can be set to delegate database access and SQL string generation to the runtime library repository instead of including within the application the more limiting code for non-delegated database access by the application. This arrangement allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. The mapping information can be used to map from objects to relational models or vice versa and generate appropriate code.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** and its description is a replication of FIG. **1** as shown and described in U.S. Pat. No. 5,857,197.

FIG. **2** is a concise flow chart depicting the operation of the system in U.S. Pat. No. 5,857,197 in a system comprising (a) an object language application program, (b) a data source manager and abstraction layer system embodiment of the U.S. Pat. No. 5,857,197 patented invention such as the CocoBase Enterprise O/R Software Tools Suite Version 4.0 (soon to be released) of Thought, Inc, San Francisco, Calif., and (c) a relational data source and JDBC driver. The operation of the system of FIG. **2** in the context of the flow chart operation will be described in detail below by referring to the FIG. **1** numbering system.

FIG. **3** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **2**, further including a number of system components or connection points for additional components. The wavy cloud that surrounds some of the system depicts the CocoBase components and the portion of the system that is called the Java environment. The additional component that are shown in FIG. **3** beyond the components described in FIG. **2** are (1) a CocoBase Object Access and Storage Adapter software module, (2) a CocoBase Object Model Repository (may be stored in a separate data source or in the system DataSource Relational Database), (3) a Plugin API (and optional caching database attached via the API) to the CocoBase Mapping Layer for caching one or more members of the group consisting of database data, query string(s), CocoBase Map(s), and data object(s), and (4) a CocoBase Dynamic Object/Relational Respository as a separate file, which could also be stored in the DataSource Relational DataBase.

FIG. **4** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **3**, further including a number of system components or connection points for additional components. The additional components shown are (1) an CocoBase Development Software module called CocoAdmin having command line and/or GUI interfaces for accessing, editing and creating new maps, for generating source code, for compiling object classes from source code, or for generating J2EE components such as EBJs from CocoBase maps, (2) one or both of a CocoBase CocoDriver Server, (3) Object Access and Storage Adapter for processing the storage of objects or object schema(s), which may access or process them in the form of objects for an object database, as a UML file for a UML modeling tool, or may convert the objects or object scheme to or from one or more of UML, XML or XMI format representations of such objects or object schema(s), (3) an optional CocoBase Object Model Repository file, which may be stored as stand alone files, in an Object or Relational data source, or an XML repository, and (4) an optional Object Modeling Tool Data File which can be exchanged between the CocoBase Tool Suite and the Object Modeling Tool in one or more directions, i.e., to the CocoBase tool suite, from the CocoBase tool suite, or both to and from the CocoBase tool suite.

FIG. **5** is a diagram depicting the comprehensive operation of the present invention as described in FIG. **4**, shown without the optional Object Modeling Tool Data File component, but further including a number of system components or connection points for additional components. The additional components are useful in a caching system or in a system for porting data from a first data source to a second data source. These additional components shown in FIG. **5** are (1) a second JDBC driver connection, and (2) a Second DataSource or Cache DataBase, where the Cache database may exist completely or in part in the system RAM, and may concurrently have a fixed storage medium component for the caching system.

FIG. **6** is a diagram depicting the comprehensive operation of the present invention that combines all of the components described in FIGS. **2**–**5**, or lists them as options within the diagram. This diagram shows a system having both a DataSource Relational DataBase and an Optional Object Database DataSource, as well as an Optional CocoBase Data Migration Tool. This migration tool can migrate and translate data between each of an Object database, a Relational database, an XML storage file, and a flat text file XMI file. An optional migration tool has the ability to migrate and translate to and from a UML format and an XMI format. Other system components have been described further in FIGS. **2**–**5**. Also, how to implement or remove various components in a system as shown in FIG. **6** is further described below in the detailed description of the invention.

FIG. **7** is a concise flow chart diagram depicting the comprehensive operation of the present invention, and including a number of variations on the basic system described in FIG. **2**. The components in FIG. **7** that have the same numbering as components in FIG. **2** will have the same description as those components as described for FIG. **2**. In general, the invention of FIG. **7**, includes (a) a new component or set of components and/or logical steps (**3**A) for caching objects, data or maps, (b) a new component or set of components and/or logical steps (**7**, **51**, **52**, and **8**) for evaluating XML information and for extracting objects, data or maps from the XML information for placement in a object (information can be in XML format) and sending the object to the Object Application **101** in response to request (**2**A), (c) an new component or set of components and/or logical steps (**4**B) for converting objects, object schema, data, meta data, or maps into one or more of an XML file or into an XMI file.

DETAILED DESCRIPTION OF THE DRAWINGS

The following detailed description may utilized terms that are defined in the "Definitions" section which immediately follows this section.

Detailed Description of FIG. 1

FIG. **1** illustrates one such embodiment involving a request **100** by an object application **101** and the processing of such request. The system according to one embodiment of the present invention comprises an object schema **200** including meta data **201** corresponding to a data store schema **300**, a first adapter **400**, and a second adapter **500**. In the preferred embodiment of the subject invention the object application **101** is a java applet "cocodemo.java" as listed in Appendix A of U.S. Pat. No. 5,857,197 ('197 patent), issued Jan. 5, 1999.

One embodiment of the '197 patent invention used a user defined object view of an underlying non-object store, i.e. the object schema **200**. The object schema **200** was accessed by an object application **101** through an abstract layer **600** which does the necessary conversions between object and non-object views. This abstraction allowed both kinds of data stores—object and non-object (e.g., relational)—to be accessed identically from at least one object application **101**.

Moreover, by applying simple object streaming, the '197 patent invention was capable of accessing objects distributed over a three tier environment using exactly the same application program interface ("API") **700** as the local two-tier environment. This wa accomplished by using access "adaptive" technology comprising the adapter abstraction layer **600**.

The adapter abstraction layer **600** of the '197 patent performs any translation work necessary for converting

objects to both object data stores **312** and non-object data stores **302**. The adapter abstraction layer **600** provided a consistent API **700** for both object and non-object data stores and enables application programmers to migrate between various object stores without application modification. The adapter abstraction layer **600** also facilitated communication with a remotely available second adapter **500** without modifying the object application **101** programming logic.

In one embodiment, the '197 patent includes the use of meta data **201** (i.e., data used to describe other data) to define how to access and convert non-object data store content **304** to objects and back. This was accomplished by paring down the non-object data store schema **300** into its various components including tables, fields, and conditions **305** in one embodiment. The paring made the creation, management, and access to the meta data **201** to be a convenient and elegant task. By using the adapter abstraction layer **600** which understands and uses the meta data **201**, the '197 patent provided an abstract view of an underlying data store(s) **302** (**312**, **322**) as an object store. The effected abstraction produced an architecture whereby the underlying data store(s) **302** (**312**) as well as multi-tier adapters (e.g., **400**, **500**, **4**XX, and **5**XX) could be interchanged without object application **101** code modification.

One embodiment of the '197 patent was implemented using the adapter technology and was therefore capable of being deployed as either a 2-tier or as a 3-tier environment transparently. The adapter abstraction layer **600**, and more specifically the first adapter **400**, communicated with a server process, more specifically the second adapter **500**, in communication with the underlying data store **302**, which transferred the requested data store content **304** to and from the first adapter **400** in communication with object application **101** client. The first adapter **400** instantiated the object **112** from more primitive data types comprising data **115**, reflecting the data store content **304** obtained from the server process, more specifically the second adapter **500**. The method of breaking down objects **102** (**112**) into the corresponding primitive types comprising data **105** (**115**) ensured successful transfers of any kind of object irrespective of object application **101** views of the object(s)' data elements. The '197 patent invention used the adapter technology to dynamically load data store access code **205** at runtime. This architecture requires objects to serialize themselves in order to be persistable or for the computer language and language runtime to be able to inspect and set an object's structure in such a way as to be able to automatically persist the object's contents. Persistable is a term of art which means an object's contents can be saved to some more permanent location such as a computer disk, and in this case refers to the storage and retrieval of the object's contents from a data store. As a result, the present invention can readily adapt to an N-tier adapter solution which accesses the data store through some intermediate data server, thereby increasing deployment and scalability options.

FIG. **1** of the '197 patent shows one embodiment of the '197 patent invention which worked in conjunction with an object schema manager **203**. The object schema manager **203** allowed data store administrators to create and maintain an object view of data store content **301** stored in the data store **302**. The object schema manager **203** could read, write, and update the meta data **201**. One embodiment of the '197 patent invention was directed to accessing non-object (e.g., relational) data stores **302**, creating meta data **201** based on data store schema **300** comprising tables, fields and clauses or conditions **305**, making up the data store **302**. One such embodiment of the '197 patent comprised a set of tools which were designed to create and maintain the meta data **201** by storing the information in a repository (such as the data store **302** itself), which information was then loaded by the second adapter at runtime. This meta data **201** then served as a map for operations available on the data store(s) **302** (**312**), for the way these operations should be performed and under what conditions.

FIG. **1** of the '197 patent further illustrated how the object schema manager **203** facilitates the creation and maintenance of the object schema **200**, thereby increasing productivity and reducing the opportunities for typographical errors, which can create preventable runtime errors. The object schema manager **203** permitted dynamic modification of object schema **200** without requiring modification or recompile of the object application **101** to ensure that the object application clients were not 'brittle' or limited to a particular database or object model as is traditionally the case in data store access applications. One embodiment of the '197 patent used the meta data **201** to construct an object view of the data store schema **300**, e.g., an object schema **200**, and used the meta data **201** to generate one or more commands **303** for accessing the data store content **304**. The deployment of meta data **201** in this way in one embodiment had the effect of using the non-object data store(s) **302** to be described and accessed as object(s) **102**, though its contents might have been more rudimentary and abstract.

One embodiment of the '197 patent stored meta data **201** in a data store **204** and dynamically loaded and executed meta data **201** at runtime. As a result, in one embodiment of the '197 patent, access code **205** corresponding to the command(s) **303** is centrally created and managed in a central repository, i.e., data store **204**. Such central repository architecture allowed the object schema(s) **200** to be re-used across objects **102** (**112**) and object applications **101** (**111**, **121**). Such an approach also allowed for any changes of one object schema **200** to be transparently reflected to any client object application **101** accessing and using the object schema **200**, without the need for object application **101** modification or recompilation. The '197 patent contemplated a notification mechanism so that data store **302** modifications resulting from requests in one location could be immediately communicated to all object applications **101** accessing that object schema **200**.

One embodiment of the '197 patent facilitated the separation of data store access code **205** (such as Structured Query Language "SQL") from the object application **101** logic. This facilitated simple and rapid application development and lowered maintenance cost for a computer system that required database access by improving the flexibility of the object application **101** and by reducing the brittleness of the object application **101** client(s). As a result, users of the invention described in the '197 patent could attain a cost savings by centralizing the task, reducing the possibility of typographical errors, installation problems, and skills availability shortages. The object stores could be created and managed by someone with data store (i.e., relational) expertise (as most modern organizations have) without requiring those individuals to also have object programming expertise.

One embodiment of the '197 patent resulted in an architecture for centrally storing and managing object schema **200** to promote re-use, effect enhanced performance, and better use of (MIS) engineering and administrative personnel skills. By way of example, storing the access code **205** within the physical confines of the non-object data store **302** in one embodiment, created a centrally manageable and maintainable system for accessing that non-object store **302**. The object schema manager **203** thus allowed for better use

of the MIS personnel's skill sets by focusing on data access in terms familiar to people maintaining their respective system. That is, one embodiment of the '197 patent allowed object application engineers to concentrate on applications code without concern for or need to learn SQL or any other native access code **205** of the data store **302**. At the same time, data store administrators could concentrate on managing and maintaining the data store(s) **302** without knowledge of application programming and without concern that any changes in the data store content **301** or data store schema **300** would adversely affect the object application(s) **101** in communication with the data store(s) **302** according to one embodiment of the '197 patent.

A central management architecture provided by one embodiment of the '197 patent allowed data store specialists to see, manage and control all access to the data store(s) **302** down to the individual access code **205** commands. It also allowed for the fine tuning and reconfiguration of those commands by the data store specialists, without involvement of object application developers. This architecture allowed the best skills to complete the respective task at hand without impacting or being impacted by a possible lack of skills that an individual for another area.

In one embodiment of the '197 patent, the object schema manager **203** allowed for the arbitrary creation and management of object schema(s) **200** via a point and click interface, thereby significantly increasing the efficiency of creating the data store access code **205**, and improving the reliability of the overall system. This embodiment also allowed completely arbitrary joining and combining of the data store content **301** as a single object view of that data, and overcame the constrained and often uselessly arbitrary or non-existent foreign key constraints of conventional mapping systems and data stores. These hard-coded links such as foreign key constraints may not be supported properly by all underlying data stores, and not be implemented by the specific customer's data store (often legacy) implementation. New foreign key constraints are often not possible on some data stores because they may not work properly with older applications if the constraints were to be created and enforced.

As a point of reference, a foreign key is generally accepted as a predefined relationship or link which the data store manages in order to ensure that records are properly related in a database. A simple example would be to have a foreign key between an order number in an order table, and an order number in a line item table which keeps the individual items belonging to that order. The foreign key in this example would be between the order numbers in both tables, which might not allow, for example, a line item record to be entered if there is no order in the order table corresponding to the order number.

A join using this embodiment of the '197 patent may occur on any arbitrary field of any table within the data store schema **300**, even if a foreign key does not exist between the fields of the two tables to be joined. Such flexibility is immeasurably useful when combined with the ability of the object schema manager **203** to access more than one data store **302** at the same time. The resulting architecture of the '197 patent allowed data store content **301** from one data store **302** to be joined with data store content **311** from another data store **312** through the object schema manager **203** without requiring any data store schema **300** changes to either system whose content is being joined. Such transparent and independent flexibility is of utmost value for the many diverse data stores currently operated by most large and small business and other entities.

One embodiment of the '197 patent provided a way to manage the data store access code **205**, requiring only an understanding of the data store **302** and the data store schema **300**, instead of requiring object programming knowledge within the modem working environment.

Once the object schema **200** was created, the object application **101** running in the application environment **103** sends a request **100** for an object **102** (possibly including an object data with lookup attributes to be matched) located in at least one data store such as an object data store **312** or a relational data store **302** in one embodiment of the subject invention. At a very high level, the request **100** was dynamically translated to a data store specific request which was then executed and results of which were parsed and new data object(s) **112** created and initialized with the results of the executed request **100**.

In one embodiment of the '197 patent, the request **100** and the accompanying object '197 patent was passed from the object application **101** to the adapter abstraction layer **600** comprising the first adapter **400**. The first adapter **400** then extracted the object attributes **103** and the object name **104** from the object **102**, and packed the object attributes **103** and the object name **104** as data **105** to be used in communication and transport layers. The first adapter **400** then communicated the data **105** and the request **100** to the second adapter **500**. The communication medium **630** between the first adapter **400** and the second adapter **500** may comprise an Internet connection, whereby the second adapter **500** is operating inside a firewall. Logical division **650** comprised a logical separation between a client, e.g., the object application **101**, and a server, e.g., the abstraction layer **600** and more specifically the second adapter **500**.

The second adapter **500** searched a meta data map **206** comprising at least one object name **114** using the object name **104** to determine whether the object name **104** existed in the meta data map **206**. If the meta data map **206** contained the object name **104**, then the second adapter **500** used the object attributes **103** and the found meta data map **201** to generate at least one command **303** for accessing the data store **302** according to the request **100**, if the command has been enabled, ensuring further security by restricting access. An example of the '197 patent contemplated using this technology for "read only" data stores over the Internet, irrespective of the user's standard privileges within the context of the data store.

The second adapter **500** then executed at least one such command **303** as a data store access client server process **700** in communication with the data store **302** using a JDBC driver to which is passed data store access code **205** and obtains the data store content **301** and an execution status **306** based on executing at least one such command **303**.

The second adapter **500** then processed the data store content **304** and the execution status **306** using meta data **201**, and packed the obtained data store content **304** and the execution status **306** as data **115**. The second adapter **500** communicated the data **115** to the first adapter **400**.

The first adapter **400** unpacked the data **115** and instantiated the object attributes **113** and the object name **114** into potentially one or more new object(s) **112** and/or status according to the request **100** and the data **115**. The first adapter **400** then communicated the request **100** and at least one such new object **112** and/or status, if object results were found, from the first adapter **400** to the application program **101**.

Several points are noted regarding preferred embodiment implementation of the '197 patent invention in the preferred embodiment. As each object(s) **102** data was being read

from the data store(s) **302** (**312**), an instance of the object's class type (or potentially some generic container object) was instantiated (in a preferred embodiment a Java Class is instantiated for each set row result returned) and initialized to the row of attributes returned in the result set. Because Java can instantiate new objects based on class name, objects could be managed by a preferred embodiment of '197 patent even though it may have been unaware of their existence or composition at compile time. For ease of implementation, each java class in order to be supported by the preferred embodiment must implement a standard interface. With compliance to this standard interface, the subject invention can call the methods necessary to initialize the data automatically, making new instances automatic and transparent to the client object application **101**.

Implementation of explicit reference handling was not required using the subject invention, and therefore is not implemented in the preferred embodiment. Unlike more rigid systems which limit the type of operation and mechanisms which will and can be used to access data stores, the subject invention allows for flexibility. This is to reflect the market reality that most organizations operate large systems in place, which should be capable of being accessed from a variety of sources in a variety of different ways. Any attempts to manage references in these mixed environments would not only prove a futile exercise, but would complicate and reduce reliability, without providing the desired functionality. As for reference handling within the object application, the Java language, which was used to implement the preferred embodiment, has language supported reference counting within its memory management and garbage collection software. It is contemplated by the subject invention, however, that specific adapters and data objects may be implemented with this architecture for specific data stores supporting appropriate reference handling management.

A preferred embodiment of the '197 Patent used an in-memory list to maintain object references, which list could be either partially or fully pre-fetched. An object application may retrieve one object, all of the objects at once, or in blocks of default, or In blocks of a user defined size. Once the objects were in the memory list, they could be iterated through using their offset in the list as their index. The '197 patent also contemplated incorporating smart vectors for automatically fetching requested Objects from the vector instead of from the data store.

A preferred embodiment of the '197 patent provided non-granular simple transaction support, although the '197 patent invention contemplated either building it's own transaction management system, or using a pass through mechanism which allowed for more sophisticated transaction management to be written by the application programmer, if the underlying system supports it. Any limitations due to the preferred embodiment of the '197 patent being implemented in the Java Driver Database Layer and JDBC or the Java DataBase Connectivity Layer were not contemplated by the invention in the '197 patent. Thus, the '197 patent contemplated using more advanced transaction management, as available in the implementing language and architecture.

When the '197 patent invention was used to access the non-object data store **302**, e.g., relational databases, the object schema **200** providing meta data **201** descriptions of how to map to the underlying data store **302** was used to generate the native access code **205** necessary to access the data store content **304**. One embodiment of the '197 patent used the object schema **200** to build an SQL statement at runtime, which was then executed to get the desired results, based on the request **100**. Because multiple statements are likely to be executed against a given class type after it has been first accessed, the '197 patent can cache the meta data for the class (name) in memory, within the second adapter **500** and saves pre-compiled versions of the access code **205** which can simply be re-executed without recompile. This approach provided improved performance characteristics for successive accesses. In the three-tier design embodiment of the subject invention, the cache was physically located on the server, and was therefore shared across all client accesses to that server, providing a performance optimization in an area where performance is most critical.

The '197 patent invention comprised the adapter abstraction layer **600** comprising a set of runtime adapters (e.g., the first adapter **400**, the second adapter **500**, the n-th adapter **4**XX, **5**XX), which can be transparently interchanged by the object application **101**. The adapters serve various functions such as direct 2-tier data access, or 3-tier or even n-tier gateway/transport. Because all adapters implement the same API **700**, they are interchangeable in the object application **101**, providing new and useful functionality, beyond implementations described in the '197 patent, without requiring object application **101** modification. Object application **101** runtime flags can instantiate entirely new adapters interfacing with radically different data store(s) **302**, without modifying object application **101**.

When data store access code, e.g. SQL, is dynamic SQL instead of pre-compiled SQL, there are typically some performance penalties to be paid for dynamic execution. One preferred embodiment of the '197 patent addresses these performance issues by pre-compiling SQL the first time a given SQL operation was executed, and by preserving the compiled statement within the second adapter **500** for future executions from the same or a different first adapter **400**. The '197 patent invention binds variables to the statement, and re-issues the generated access code with the new SQL. As a result, the penalty as to dynamic execution became a slight overhead the first time an object **102** of a given type was requested from the abstraction layer **600**. Past this first point the performance characteristics should be the same as from any other embedded SQL client, perhaps even better because the amount of information to be processed and transferred to the server accessing the data store **302** was likely to be less using the architecture of the '197 patent than one which embeds SQL within a client object application and ships it with every request, and then must process the results inline, and often unnecessary descriptive data that is returned, and which is unnecessary in a system which is already preprogrammed with and understanding of the composition of the return values.

Because the first adapter **300** packed object request(s) **100** as a single API **700** call returning a single complex result (building the resulting data objects **112** transparently in the client from the returned complex data **115** packet), the '197 patent invention reduces network latency (which is essential to reduce over the Internet and other io-bound and high latency networks) and increases the performance and scalability of the involved clients and servers. The '197 patent invention also reduces the quantity of extraneous information which is always returned to client object applications by other prior art systems such as the JDBC type-3 distributed database access layer architecture upon which other common Java Database Products are currently built. As a result, not only did the '197 patent architecture obtain tremendous performance gains by reducing the latency of multiple network roundtrips, it also reduced the quantity of data and therefore increased the number of clients it could support, since each request/response required less I/O. Compression

and data duplication reduction techniques were contemplated by in the '197 patent to further increase performance.

New adapters, e.g., **4**XX, **5**XX, could be added to the adapter abstraction layer **600** dynamically, even at runtime, as long as they complied with the API **700**. This allowed data store(s) **302** (**312**) to have custom adapters built for them (such as object data stores or other relational and/or non-relational data sources) which comply with the API **700** of the subject invention and are loadable in the same way as the adapter abstraction layer **600** described in the '197 patent.

Detailed Description of FIG. **2**

FIG. **2** is a concise flow chart depicting the operation of a computer system having an implementation of the "System and Method for Accessing Data Stores as Objects" described in U.S. Pat. No. 5,857,197. The system of FIG. **2** comprises three basic components (a) an object language application program, (b) a data source manager and abstraction layer system embodiment of the U.S. Pat. No. 5,857,197 patented invention (such as the soon to be released CocoBase Enterprise O/R Software Tools Suite Version 4.0 of Thought, Inc, San Francisco, Calif.), (c) a relational data source and corresponding JDBC driver, and (d) an optional Maps Files Storage facility. The operation of the system of FIG. **2** in the context of the flow chart operation will be described in detail below by referring to the FIG. **1** numbering system that was described above.

FIG. **2** starts at position **1**A (Start/End) with a request step **2**A from an object application (**101** in FIG. **1**), which calls the CocoBase DataSource Manager and Abstraction layer (hereafter CocoBase system) and delegates the database accessing to CocoBase. The logical loop that the CocoBase system performs begins in response to the request step (**2**A) and first performs the Extract Object Name step (**4**). The CocoBase system loop continues onward to the Evaluate Object Properties step (**5**) and its associated components where (i) object properties are evaluated (**5**), (ii) object to relational maps are accessed, created, or updated by **3**B (and this may involve retrieving maps from the system DataStore **3**C or from a separate Optional Maps Files Storage facility (**1**C)), and (iii) database commands necessary for data retrieval (including SQL strings) are generated at step **2**B under the control of the Execution Status Monitor (**1**B), before the CocoBase system continues onward to the Obtain Data step (**6**). In step **6**, the CocoBase system in coordination with components **1**B, **2**B and **3**B, passes SQL strings or other DataBase Access Statements from **2**B to the JDBC Driver (**2**C), which obtains data from the system DataStore (**3**C) and passes that data to the CocoBase system. Then the CocoBase system places the obtained data in an object (Step **6**A, Place Data in Object and sends the Data Object (Step **7**A, Send Data Object) to the Requestor Object Application (back to Step **2**A position) and the delegation to the CocoBase system by the Object Application ends (back to Step **1**A).

In the paragraph above, the general functions of the system of FIG. **2** have been explained. In the paragraphs below are more detailed descriptions for the (un-illustrated) CocoBase system components that may be involved in each step of the flow chart (or CocoBase components associated with that step as illustrated in FIG. **2**) while using FIG. **1** reference numerals to better describe parts and functions of those CocoBase system components.

Please note that the description of FIG. **2** in the paragraph above and in the paragraphs below describe the CocoBase system as accessing a JDBC driver. This is a simplistic view of the system. Whenever an Application Server is utilized to run an object application and to coordinate with CocoBase and "XA" component often "wrappers" a JDBC driver and all database accesses are passed through the XA component. This more complex architecture is designed to monitor all data source activities in order to provide reliable transaction monitoring. Even with such a complex architecture the principles described above are valid and the application code still "delegates" all database accesses to the CocoBase system and does not directly access the JDBC driver for a given data source.

At step **1**A, the Object Application (**101**) may contain two referenced parts (a) Object Schema (**200**) and (b) Meta Data (**201**), and its Request (**100**) for data at Step **2**A, which is delegated to the CocoBase system.

At Steps **4** and **5**, the Abstraction Layer Adapter **1** (**400**) may involve the Object Schema Manager (**203**), Object Schema (**200**), object data store contents (**301**) of DataStore (**302**) or in a separate external mapping repository storage facility (e.g., Optional Maps Files Storage (**1**C)) to obtain the Object Name (**104**) and Object Attributes (**103**) for objects (**102**) or (**112**) as data (**105**) to be used by the CocoBase system. Particularly at Step **5**, the data (**105**) may be passed (optionally across a logical division **650**) to a second Abstraction Layer adapter (**500**) to an abstraction layer portion **600**. The second Abstraction Layer adapter (**500**) checks the meta data maps (**201** or **206**), which may be an Object/Relational map. The adapter (**500**) uses the object attributes (**103**) and the data map (**201** or **206**) to generate at least one command (**303**) such as an SQL string or other DataBase Access Statements (**2**B) under control of the Execution Status Monitor (**1**B) of adapter **500** for accessing a database, e.g., relational DataStore (**302**/**3**C).

The second adapter **500**, at Step **6** of FIG. **2**, then executes at least one command **303** as part of a data store access client server process **700** in communication with a JDBC driver (item **2**C in FIG. **2**) which converts the at least one command into a format that is understood by the data store **302** (item **3**C in FIG. **2**). In **2**C of FIG. **2** the JDBC driver communicates with the data store item **3**C by using data store access code **205** and obtains at least one item of data store content **301** and an execution status **306** based on executing at least one such command **303**. The client server process **700** passes this data store content **301** and execution status **306** back to item **6** (Obtain Data) of the CocoBase system which coordinates with both the client server process **700** of CocoBase and the Execution Status Monitor component of FIG. **2** (item **1**B) to provide this information to the second adaptor **500** of the CocoBase system.

The second adapter **500** then processes both the obtained data store content **304** (collection of at least one item of data store content **301**) that corresponds to the request and the execution status **306** by using meta data **201** (of **3**B in FIG. **2**), and packs the obtained data store content **304** and the execution status **306** as data **115**. The second adapter **500** of Step **6** in FIG. **2** communicates the data **115** to the first adapter **400** of Step **6**A in FIG. **2**.

In Step **6**A of FIG. **2**, the first adapter **400** unpacks the data **115** of Step **6**, and instantiates a set of object attributes **113** and the object name **114** (object attributes and object name are obtained in Steps **4**–**6** of FIG. **2**, see description above) into potentially one or more new object(s) **112** (see FIG. **2**, Step **6**A "Place Data in Object) and/or status according to the request **100** (the original request from Object Application **101** for data from Step **2**A of FIG. **2**) and the data **115**.

In Step **7**A of FIG. **2**, the first adapter **400** of the CocoBase system then communicates the request **100** and at least one such new object **112** and/or status, if object results were

found, from the first adapter **400** to the application program **101** request of Step **2**A in FIG. **2**. The Object Application **101** at **2**A indicates receipt of the information to the CocoBase system and at Step **1**A the Object Application completes and ends the data request process.

Detailed Description of FIGS. **3**–**6**

FIGS. **3**–**6** illustrate more complex variations of the CocoBase system that is described in detail above with respect to FIG. **2**. Different components and features are added to the basic system of FIG. **2** to provide features and functions that are described below in detail in the detailed description of the invention. See the description below for instructions on how to add or remove such additional CocoBase system components and their features to individual computer systems. Such details are provided below contextually for the individual components.

Detailed Description of FIG. **7**

As indicated above, FIG. **7** is a concise flow chart diagram depicting the comprehensive operation of the present invention, and including a number of variations on the basic system described in FIG. **2**. The components in FIG. **7** that have the same numbering as components in FIG. **2** will have the same description as those components as described for FIG. **2**. In general, the invention of FIG. **7**, includes (a) a new component or set of components and/or logical steps (**3**A) for caching objects, data or maps, (b) a new component or set of components and/or logical steps (**7**, **51**, **52**, and **8**) for evaluating XML information and for extracting objects, data or maps from the XML information for placement in a object (information can be in XML format) and sending the object to the Object Application **101** in response to request (**2**A), (c) an new component or set of components and/or logical steps (**4**B) for converting objects, object schema, data, meta data, or maps into one or more of an XML file or into an XMI file.

These additional components and features of FIG. **7** that are added beyond the basic system of FIG. **2** relate to providing features and functions that are described more fully below in the detailed description of the invention. See the description below for instructions on how to add or remove such additional CocoBase system components and their features to individual computer systems. Such details are provided below contextually as relates to the individual components of FIG. **7**.

DEFINITIONS

For the purposes of the present application, the following definitions are given as a meaning for terms used herein throughout this application to avoid any confusion with possible multiple meanings for such terms. Other terms used herein have meanings that are well recognized in the art, and their meanings will be clear from the context in which they are used in this application.

A "module" in the computer programming context is an organized set of computer code designed to act on some externally passed in data, where everything needed for that action is passed in to the module.

An "object" in the object oriented programming context is an organized set of encapsulated programming code designed to act on itself at the request of some external system, which system may pass in some additional information to the object when it delegates a task to the object in that request.

A "composite object", or an object programming "component", in the object programming context each refer to an object comprising a complex and organized set of objects that are encapsulated to form the composite object. The two terms "composite object" and "component" (e.g., a J2EE component such as an EJB, Enterprise Java Bean) may be utilized in an interchangeable manner in referring to the same type of logical constructs and concepts.

A "delegation" in the object oriented programming context is where an object or a programming application permits another simple object, set of objects, composite object, or a module to perform an action by simply requesting the action from the delegated simple object, set of objects, composite object or a module.

A "non-delegation" database access in the object oriented programming context is where an object or a programming application has specific code imbedded in the application which directly controls database calls and the generation of SQL strings, wherein the imbedded code is specifically tailored for accessing a particular database or for a specific database schema.

A "user interface" for an object oriented application, such as a Java Server Page (JSP), a Swing GUI, and the like, refers to a software component or module that provides a feature for a user that will permit the user to interact with an object or programming application in some way, such as the interactions of finding, selecting, inserting, updating and deleting data in a database.

A "library" is a set of definitions, data, objects or programming modules that may be accessed by a computer programming application to obtain information or to delegate tasks.

A "repository" in the object programming context and in the context of this application is a special set of libraries that may include, among other things, items related to object to object mapping, object to relational mapping and database accessing information, database parameters, optimized database access routines, and the like. A repository may be a single file or may be a set of files. The format of items in a repository may vary widely according to the desire of a computer programmer user or developer and may be in one or more of formats such as simple text, XML, XMI, UML, JDBC, source code, compiled code, and the like.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is based in part on U.S. Pat. No. 5,857,197, (incorporated herein by reference), and provides a mapping system for handling data requested by an object software application model in a manner that is compatible with relational data stores. A dynamic repository-based mapping system is used. The system does not put all of the data access code in java objects, for example metadata (data about java objects including complex java objects that have relationships with other java objects) does not need to be stored in a java object. Instead, the mapping information related to object definitions and some metadata can be placed in a separate structure that is independently stored. This allows the mapping information and associated metadata to be easily accessed, changed and used to convert thousands of lines of code in a data object, as needed. Mapping information can be used to map to and from objects to relational models, objects to objects, object to COBAL or vice versa, and object to XML and the like.

Certain of the embodiments of the present invention are embodied in a suite of products by Thought, Inc., referred to as "CocoBase". Aspects of the CocoBase technology are also described in U.S. Pat. No. 5,857,197, supra. Also, a

number of details of the system of the present invention are provided now as state of the art in the documents that have been posted on the website of Thought, Inc. or are otherwise available on the internet as publications.

In one embodiment, the mapping information, rules, or metadata can be maintained in a human-readable format and can be interpreted by an application to apply the rules to data objects. This information can be stored directly in the data source or can be stored in a separate file. In either case, this information is called a "repository". This repository provides an easily maintainable and transportable format for the mapping rules. The rules and java object relationships can be updated, or otherwise changed, dynamically even while other rules or object relationships in the repository are being used to perform conversions. The rules and relationship definitions can be embedded in standard formats such as in Java code, e-mailed, sent as plain text, etc. This allows for flexible transfer and proliferation of the rules to other sites where the rules can be used to access objects (which can be bundled with the rules), used to derive other rules, etc. In a particularly preferred embodiment, this repository is in an XML (extensible markup language) format, and the object model definitions can also be present in an XMI (XML metadata interchange) format or can be easily exported to an XMI file format.

A more preferred embodiment is such a system and software as provided by an updated CocoBase Enterprise Object/Relation Software package (hereafter CocoBase), which includes the software tool CocoAdmine. Such updated software package, is available, or is available shortly, from Thought, Inc., San Francisco, Calif. An embodiment of the invention using this package is described below.

Exporting Maps as XML Repositories in CocoBase

CocoAdmin provides a mechanism for the export of maps defined against a database into a modifiable XML format. This facility allows a system user to select an existing map(s) to be exported, and to also specify the filename of the XML document to be created. The resulting XML document is written to a file with the specified filename using an XML template named coco.dtd that is ordinarily located in the thought\cocodemo3tier31\demos\resources directory.

Once the XML file is written, it can be edited using a standard text editor or XML editor, and can be modified to reflect map customization requirements. The following discussion relates to specific features of the generated XML file.

The basic repository format has CBObject (CocoBase object class) definitions that reflect the select, insert, update, delete and call related map definitions for a CocoBase map. Each of those operations further consists of tables, fields and clauses that may exist to specify how the object is mapped to and from the data source.

XML files contain a DTD (document type definition) entry which describes the structure of the tags and data contained in the file. The DTD only checks for the existence of required elements, but doesn't check the contents of those elements. Given this desired degree of freedom, it is possible to enter invalid map information which will not be processed properly by the CocoBase runtime and care should be exercised to avoid improper editing. In general, modifications should be restricted to the schema, table, and field name variables in the Tables and Fields entries. These variables may require different values when they are exported from one database instance, and imported into another.

To begin exporting a set of maps, select File->Export XML Map Repository from the CocoAdmin pull-down menu. Multiple maps can be selected by holding down either the <Shift> or <Control> when selecting the connection and map(s) to export.

When Next>is pressed in the dialog box provided, the list of selected maps will be presented in a list and the window will prompt for a final acknowledgement of export. The XML repository filename including the directory path can be specified in this window. If the directory path is not specified, the XML repository will be written to the thought\cocodemos3tier31\demosdirectory. When the Export XML>button in the dialog is pressed, CocoBase creates the XML file in the specified directory. After the document has been written to the disk a dialog appears which acknowledges that the repository has been created.

Importing Maps from XML Repositories Using CocoBase

CocoAdmin provides a mechanism for importing XML based CocoBase map definitions, or for importing an XMI object model definitions and then generating corresponding XML based CocoBase map definitions. Using the XML syntax defined in the thought\cocodemo3tier31\demos\resources\coco.dtd template file, CocoAdmin can import maps previously defined and exported from a different database or from a different object instance. The XML files generated from CocoAdmin will be validated against the DTD, and a basic syntax check will occur. No in depth syntax checking will occur because of the flexibility allowed in the system. When an import of an XML map definition occurs, it is in relation to an open database connection.

To begin importing an XML map definition from the CocoAdmin GUI select File->Import XML Map Repository from the pull-down menu or Import XML Map Repository from the popup menu in the main CocoAdmin dialog. Import an XML based map definition by selecting the database connection into which the import is to occur, and click the Browse button to find the XML repository file to be opened. After a file is selected the dialog button Load XML Document is clicked and the selected XML filename appears in the initial Import dialog. After the Next>button is pressed, the selected maps are compared with those already in the database. If a newer version of a map already exists in the database, by default a flag is set to retain the newer version. If the maps being imported are newer than the ones already in the database, then by default, a flag is set to import each of the newer ones. When the comparison operation is completed the user can override any flags before the task is initiated. For example, if the XML maps are older than the versions already in the database, this condition causes the Already Exists? flag to be checked and the XML map definitions will not be imported unless the user overrides this flag by checking the Import/Overwrite? box.

When a user clicks the Import XML Repository button as described above, maps marked for import will automatically be integrated into the list of maps in the CocoBase repository for that database. If the imported maps do not reflect the physical structure of the underlying database, they may need to be manually edited by the map editor before using them with an application. Also, XML format maps can be edited before they are imported. For example, an XML editor or any standard text editor can be used for this operation. Most common edits consist of changing column names, table names, and less frequently, the schema names of the tables and the fields that the map may access.

The CocoBase Programmer's Guide available at the www.thoughtinc.com website provides further information about how to work with XML format maps and CocoBase, which published document is incorporated herein by reference. This document also explains how to set CocoBase (or CocoAdmin of CocoBase) and its runtime modules to cache maps that an application will be using in order to speed up user access. Other user hints and instructions are provided therein. For example, one URL option in CocoBase is to specify an XML repository for the maps that an application will be using. Instead of using the database repository, the runtime can use an XML repository exclusively. This technique allows CocoBase to execute against a production database that cannot be modified with map definition tables. Optional mapping server plug-ins are also described.

Data source maps according to the present invention may be utilized to generate application programming code, such as Java source code. For example, the CocoAdmin tool will allow a user to generate Java for a range of targets including most commercial Application Servers and Enterprise Java Bean Servers. A complete list of pre-configured targets can be viewed from the Connections window in the Generate Java wizard of CocoAdmin. Any Java class can readily be used with CocoBase, whether or not it was generated by the CocoAdmin tool. However, there may be significant advantages that occur from using the code generation facilities of CocoBase to quickly generate your Java classes. For example, the disadvantages inherent with data source specific non-delegation database access code can be avoided by generating code, which delegates the database access functions to CocoBase runtime libraries. Only CocoBase is believed to utilize this type of code that results in dynamic O/R mapping system capabilities.

The CocoAdmin tool can use a combination of the map definition and database foreign key relationships (if any exist) to define how the Java class is going to be generated. The Generate Java wizard of CocoAdmin can be started in one of four ways to generate Java code from a CocoBase map.

1. Clicking the on the coffee pot icon located on the CocoAdmin tool bar.
2. Selecting File->Generate Java Code from the CocoAdmin pull down menu.
3. Selecting Generate Java from Existing Map from the launch pad after a connection has been made to a database repository, then clicking Launch.
4. Right clicking in the CocoAdmin main window and selecting Generate Java Code from Map from the pop-up menu.

From the connections directory tree in the Generate Java wizard, select the map from which a Java class will be generated. From this window, the CocoAdmin user can also set flags to automatically generate code that does one or more of the following:

Supports the transaction object (check TransObj).

Uses existing foreign keys to automatically determine object relationships (check ForeignKeys).

Uses the CBDrop, CBProp or both of CocoBase persistence interfaces (check CBProp and/or CBDrop).

The CocoAdmin user can also (optionally) enter in a package prefix at this point. Java classes are typically organized into packages, which are stored in a common directory. The package prefix (entered at the Pkg Prefix: prompt) will then refer to the directory in which the Java code will be stored.

The Code Generation Template drop-down list of the CocoAdmin code generation wizards allows a CocoAdmin user to select a target for the classes (Java code) that will be generated for a member of the comprehensive list of targets. The following target categories are supported for most commercially available application servers and EJB servers.

Java for distributed objects and application server applications.

Enterprise Java Beans with Bean Managed Persistence (BMP).

Enterprise Java Beans with Container Managed Persistence (CMP).

The Generic EJB Entity Bean CMP—All Parts option from the drop down of CocoAdmin will generate a completely standard CMP Bean, which can be installed using any vendor supplied CMP installer. CMP concepts for EJBs are discussed in a variety of documents published in this field, including in the appropriate section of the CocoBase Programmers Guide. In order to take full advantage of the CocoBase O/R mapping features, such as delegated database access and the like, a user must install the generic CMP with the CocoBase CMP installer tool, which is an optional component of the CocoBase software package. This tool will configure the CMP EJB to run in most commercially available EJB servers that support container managed persistence EJBs and coordinate the database access with the CocoBase runtime libraries.

Clicking Next>in the Generic EJB Entity Bean CMP—All Parts drop down wizards of CocoAdmin will present a user with a list of attributes which comprise the maps and any references (e.g. foreign keys) to other maps that may be navigated by CocoBase. For example, in a e-commerce shopping cart example, a Customer map, which is generated against a selected relational database connection, might be selected, and the PkgName: field might contain the name testpkg so the resulting Java code will be generated to a package and directory named testpkg. In a subsequent pop-up dialog, the user can add a foreign key reference by clicking on Insert Attribute and filling in the foreign key attribute in the inserted row. Such references can be added automatically for any operation which contains a join across tables of different maps. This topic is covered in more detail in the CocoBase Programmer's Guide.

Customizing Attributes to be Generated in Java Code by CocoAdmin

Attributes that are to be generated into the Java code by CocoAdmin can be customized through the code generation screen of CocoAdmin, which is displayed when a map is selected from the wizard connections tree and the Next>button is pressed.

For each fixed Map Attribute Label in the leftmost column of the table corresponding to a selected map, the corresponding Java Attribute Name and Field Type are editable. When the Field Type field item is placed in edit mode, the CocoAdmin user can select one of the available data types from the drop-down list. These relational data types will be mapped to a corresponding Java type when the code is generated. A relational to Java conversion table can be found in CocoBase Programmer's Guide, but any functional conversion table may be used. If foreign key references are present, each entry of the reference is also editable.

In general, the attributes for Java code to be generated should not be customized here unless a foreign key relationship needs to be modeled that isn't described by the database foreign keys or that isn't described correctly. Database foreign keys are automatically included in the map if the ForeignKeys checkbox in the wizard connections

window is checked. Editing a map to include foreign key relationships is covered in the CocoBase Programmer's Guide.

A CocoAdmin user can specify a key for the map for the purposes of code generation by checking the box under the Key column. The attribute corresponding to the checked key box will be used as a key for the map and more than one box can be checked. For non-EJB applications, it is not necessary to specify a key, but keys can be useful in applications such as object caching and should contain unique values. Because EJBs require a unique key (for example, see the EJB spec. 1.0 from Sun Microsystems), CocoAdmin will not let a user generate code for an EJB without first specifying a key.

Customizing an Output Filename for Generated Java Code

When a map is generated into Java code by CocoAdmin, the output class name is specified manually by entering it in the File name field. If no filename is entered, the user will receive an error message when the user attempts to generate the Java code. Generally the user should adhere to Java naming conventions when naming package components. Package components should be saved in the PackageName directory and Java class files should have the form Classname or ClassName.java when entered into this field.

A CocoAdmin user can specify an existing output directory for the generated class (generated code) from the Look in: drop-down dialog. If from a previous window, the user specified a package prefix for the source code, then a subdirectory matching the prefix name can automatically be created in the specified output directory, if one does not already exist. Further information on naming and naming customization is presented in sections of the CocoBase Programmer's Guide.

Generating Java/EJB Code Using CocoBase

Once a filename has been entered as described above, a CocoAdmin user can generate the Java code by clicking the Generate button. When the Java Class or EJB has been generated, a dialog box indicating a successful generation will be displayed. If a user attempts to generate an EJB without specifying a key attribute, an code generation exception key must be specified! error message is received. Likewise an error message may be received if a file already exists with the filename that was specified by the user for code generation.

Prior to code generation maps of CocoBase may be customized to add relationships between fields, objects, and beans by creating links. Relationships between objects such as 1 to 1, 1 to many, and many to many may be set. Among other relationships, parent child class relationships are supported. Code may be generated which reflects such relationships. Also, during the code generation step relationships may be added using wizards and interfaces of CocoAdmin.

Link information can be dynamically entered and edited with a CocoAdmin dialog, or it can be loaded from the resources\coconavLink.properties file of CocoBase if code regeneration is going to occur often with the same class, and if keying in virtual foreign keys isn't practical. The database foreign keys are auto-detected at a table level, and can be edited using a CocoAdmin dialog if the Map name to be used is different from the default one generated using a table name.

See Appendix A and B for links and maps related to Inheritance and Cartesian models.

Customizing Generated Code through CocoBase Templates

The later versions of CocoBase Enterprise all have the ability to customize code generation through the use of multiple code generation templates. Templates are available from the tool through the cocoadmin.properties configuration file. Each of the GENTEMPLATES name values found in a configuration file are comma delimited and specify the name of the code generation template to be used. The comma delimited file can be viewed using an ASCII text editor, for example. The "\" character which appears at the end of each line indicates a line continuation and is only relevant to the CocoAdmin tool. Name values present in this file can also describe a set of templates that will be processed simultaneously. The code generation list that appears in the tool by default includes, among other things, all of the major Java Application Server types, production of generic CMP, generate Java classes, generate JSP, and the like. See the current www.thoughtinc.com website for a more complete listing. The default Java object of the listing is a standard Java instance that implements the CocoBase interfaces. If a GemstoneJ EntityBean using Proxy objects is selected, for example, CocoAdmin will generate all of the GemstoneJ files necessary for a Bean Managed Persistent EJB Object. Such an "All Parts" template reference will actually cause several files to be generated. Such a selection would have a cocoadmin.properties entry that specifies which template files are to be processed, and in what order to process them. Such a file might be found in the thought\cocodemos3tier31\demos\resources directory of a particular version of the installed CocoBase software package. In one embodiment of the invention, when a particular template is picked any file prefix/suffix values to be appended to the filename would also be prefixed or appended to the map name automatically.

Mapping from one Data Source Format to Another Data Source Format

In another embodiment the invention provides a system for mapping from a first database format to a second database format, or from one database to another database of the same type, as a way for transferring data or synchronizing data sources. The system includes: data in the first database format stored in the system; rules for translating from the first format to the second format stored as a separate structure from the data; and means for applying the rules to the data to obtain the second format. This system may include a means for transferring the data from a first data source to a second data source. In one embodiment this system includes a simple computer program in a computer language such as Java, which reads the data from the first database using a repository map that may optionally be cached in the computer memory of the system and then stores the accessed data to a second database using the same or a different repository map.

An example of the structure for such a system and a short description/outline for a computer program adapted to such a system that is capable of transferring data between a first data source and a second data source might be described as follows:

First, let us assume two databases: a first database (B**1**) and a second database (B**2**) wherein B**1** and B**2** have the different corresponding data map schemas (S**1**) and (S**2**), respectively, and wherein S**1** and S**2** have the two corre-

sponding schema repositories (R1 and R2) that include maps defining the structure (schema) of the database, Java object information (Java object model and any relationships between Java objects of the Java object model, or defining both the database and Java object information.

M1 is a map (or maps) defining the database schema S1 of R1 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B1, and

M2 is a map (or maps) defining the database schema S2 of R2 including map (or maps) with definitions for relationships between Java objects corresponding to the data of B2.

The sample program outline,

(i) open a first system connection to access R1 and read M1

(ii) open a second system connection to access R2 and read M2

(iii) have a Java application create a Java object (Obj-1) in memory that can access the data of B1

(iv) allow the Java application to connect to B2 and access B2, or have the Java application create a Java object (Obj-2) in memory that can access the Obj-1 and also access B2,

(v) open B1 and B2

(vi) have Obj-1 retrieve data from B1 and place data in Obj-1

(vii) optionally have Obj-2 obtain data from Obj-1

(viii) have the Java application access B2 (optionally through Obj-2) and store the data of Obj-1 or Obj-2 in B2

(ix) clear Obj-1 and possibly Obj-2, or create a new instance of Obj-1 and possibly of Obj-2

(x) repeat steps (vi)–(viii) until all of the data is transferred from B1 to B2

(xi) close B1 and B2 and any other open files, and

(xii) end program.

In one preferred embodiment, step (vii) as described above involves having data retrieved by Obj-1 and then converted to an XML format data file which is then forwarded to a different location (and optionally simultaneously to a different user) on a distributed network, whereupon an existing or new instance of Obj-2 is populated with data from the forwarded XML format data file and Obj-2 then stores the data in B2. The XML format data file may be simply of a Java object, the data than can be stored in a Java object, or both the Java object and its data.

The above procedures implement a dynamic mapping layer in an exchange format such as XML that can directly exchange data from two data sources in a more dynamically controllable fashion to each other for the purposes of data import, export and exchange without the requirement of an intermediate third object. In such a model the developer would not be constrained by the structure of the database to which or from where the data is being transferred.

Also, the above example is an example of object to "DataSource" mapping. Data may be mapped to (or from) object and relational databases and may also be stored in XML. Also, XML stored maps can be utilized by CocoAdmin as a resource from which to generate object code for user applications. In one implementation, two system users (same or different system) using a single (or several major recognized) XML standard(s), can export or import (as XML files) either or both of the XML map definitions (relationships of data) and data (dataset) of the database. This would permit a user to distribute parts or all of a dataset (and/or maps) to a distributed network, or to multiple network users in useable XML format files. Such a phenomenon may be referred to generically as O/X mapping (object to XML) or R/X (relational to XML) in addition to O/R (object to relational mapping).

In one embodiment, O/X mapping step would permit importation or exportation of data (that was converted to XML) from XML to and from any business object. For example, data from any business implementation object could be exported as XML along with its relationships as defined by the mapping tool. This would implement a more flexible and dynamic mapping facility for taking XML datasets and using them to populate object instances other than the original ones that may have created the data. One good example of such an implementation would be to take a first XML document that is presented as a user screen interface to a dataset and utilize the information (dataset) populating the first XML document to seamless populate a second XML document (datasource to datasource via a XML (or another exchange format similar to XML) translation step.

There are many ways to utilize the above above implementations as part of either uni-directional or bi-directional mapping. Intranets and e-commerce applications can both take advantage of these important features. There are distinctive advantages when using a dynamic mapping layer where a java object provides translation by mapping such objects of a first data source (relational or object database) and also mapping such object to an XML or other second format data source. This allows more developer control as to how datasets are exchanged, filtered and/or validated between a first data source and a second data source.

One business use would be for a first party to use XML as a data export document to send information to a second party (e.g., items offered for sale) and the second party would import that information in order to generate a second XML data document (e.g., purchase order) that the first party can import. This would provide a much needed exchange format for the newly developing business to business market that might be able to rely on standardized XML data formats and avoid problems caused by object programming models that may vary widely from company to company.

Such a dynamic mapping facility would provide a critical missing facility to easily translate data models and object model into a format that can be readily used by business to business companies, or by the same company for department to department interactions.

Documents describing the use of CocoBase products that do mapping of data sources other than relational databases have been published, and such products may be used in conjunction with the above embodiments. Examples of mapping include, mapping from object to object databases and from COBAL (mainframe computers of businesses often use this language data source) to object. See the IBM "white paper" on the topic of COBAL to object mapping (may be downloaded from URL http://www.thoughtinc.com/websphere.html).

In one embodiment of the system of the present invention a translation layer translates between an object application (or a potential object application, i.e. an object model) to at least one relational database which includes data entries organized as tables and records. Examples are database architectures supported by companies such as Oracle, Sybase, Informix, etc. Such an organization is well-suited for manipulation by relational query languages such as SQL. However, the traditional relational database organization is not ideally suited for manipulation by an object-based system. Some tools, such as Java Database Connectivity (JDBC) exist to achieve some degree of indirect relational to

object mapping by accommodating some differences between an object model or application and the structure of the relational database. However, these tools also have drawback in that they are often not scalable, require extensive manual recoding for different object applications and are complex to use. Even tools (often mistakenly referred to as object to relational, i.e., O/R tools) that generate JDBC code (instead of generating a pure SQL string that a JDBC driver can utilize to subsequently create a pure SQL statement or query bundled in a JDBC format) are limited since they are often application specific, database specific, or specific to both.

In one preferred embodiment, a translation or abstract layer communicates with at least one JDBC (relational database driver) and at least one primitive Extended Java Bean (EJB) construct. The function of such a translation layer (generally called the O/R layer in CocoBase documentation, for example) is to translate object-based queries for the data into queries that JDBC can, translate into queries for a relational database. In a preferred embodiment, the translation layer can generate an SQL string (or strings) based upon the object-based queries, which can be passed to at least one JDBC, which JDBC can then generate an SQL statement from the SQL string. Similarly, the abstract layer accepts results from the queries and provides them to one or more of the EJB constructs in a suitable object format. The existence of JDBC is not necessary for all implementations of the invention. Different types of databases, data models, computation architectures, and the like can be used within the basic principle of the present invention. This mapping can be one-to-one, many-to-one, many-to-many, or any variation.

Another preferred embodiment of the present invention allows for mapping tables to be plain text, or text-based XML repository files. This not only allows the maps to be human readable and editable with standard editions, email programs, web browsers, etc., but allows for easy transportability and delivery of the maps.

Another feature of the maps is that they can be dynamically loaded into a computer system to provide for new dynamic mapping in a system that can that can continue to run while the new maps are loaded and executed. As described supra, the use of the translation maps provides advantages in organization, standardization of interfaces, efficiency of design, compatibility, scalability, portability and other advantages. The system of the '197 patent provided an entire set of tools to simplify and improve an operators ability to manipulate the maps without requiring high levels of programming knowledge or database query language familiarity.

In part, the present invention is based upon a translation operation derived from the operation of U.S. Pat. No. 5,857,197, and depicted by the flow chart at FIG. **1**. At step **1**(*a*) the application software is operating, and reaches a point where data must be accessed to continue the operation. For purposes of example, the application software is object oriented, and is being run by a customer at a customer location. However, the application software need not always be limited to object orientation. Rather, other types of software models can be facilitated by the present invention to be described herein.

Once the application software reaches a point that data is needed, for example, a production database or data store, which is not at the site of the application or controlled by the application user, then a request **2**(*a*) is made for that data.

As described in great detail in U.S. Pat. No. 5,857,197, when an object application makes a request for data such as **2**(*a*), then certain object information is sent to a location whereby the necessary data can be accessed. Such activities may be coordinated or regulated by a transaction coordinator facility of an object application server.

The object information provided the object application is received in an abstract layer, also known as a translation layer, at step **4**. At this point, an analysis takes place to extract certain information regarding the object sent by the object application. This information is used to select and manipulate a particular metadata maps provided by a data manager, which handles the data store containing any information required by the object application.

Operating entirely independent of the application software is the data manager. The data manager carries out an analysis of the arrangement of data in a data store controlled by the data manager at step **1**(*b*). Depending upon the size of the data manager and the data store, any number of analysis can be carried out to help facilitate access by parties interested in the data held by the data store.

The result of this analysis is the development of a metadata map or maps based upon generated object characteristics of the arrangement of the data store. Depending upon the size and arrangement of the data store, any number of metadata maps can be generated. The number of generated object characteristics, which would be the basis of the metadata maps depends upon the data manager operator, and his/her response to the requirements and characteristics of those seeking access to the data store.

At step **3** a metadata map or metadata maps, based upon the object characteristics of the arrangement of the data store, are generated. The number and characteristics of these maps depend upon the characteristics of the arrangement of the data store, and the responsiveness of the data manager operator to the various types of demands for data by object applications. Each metadata map provides the necessary means to obtain data or particular types of data from the data store. It should be noted that in many cases there are many object characteristics can be derived from a relational data store. The data related to particular derived object characteristics that can be accessible using a metadata map generated for a selected group of object characteristics.

The operation of the object application and the management of the relational data store come together at step **5**. At this point, an evaluation is made between the object information provided from the object application and object characteristics associated with the metadata map available in the data manager. The evaluation results in a selection of at least one metadata map appropriate for the object characteristics sent from the application. The selected metadata map is used to navigate through the data store to obtain data that is desired for the object application at step **6**. It should be noted that more than one metadata map can be used to obtain data from the data store.

Once the data is obtained, it is sent back to the requesting object application at step **7**. It should be noted that as described in U.S. Pat. No. 5,875,197, step **6** of obtaining data and step **7** of sending data are constituted by a much greater number of sub-steps. These include: generating demands from the metadata map; executing the commands used by the metadata map; obtaining data from the data store in response to the command; processing the data access from the data store using the metadata map; packing and manipulating the data for transmittal; unpacking the data; instantiating objects using the accessed data and returning the instantiated objects to the object application for further processing. It should be noted that the results of the overall

data accessing process may contain many rows of data which can be used to instantiate a plurality of objects.

An important purpose of the aforementioned system is to achieve a high degree of flexibility with dynamic mapping to objects as they occur in object applications. Accordingly, it may become necessary to handle a plurality of objects at any one time, and obtain the necessary data to instantiate or otherwise satisfy requirements of each of the objects addressed in the object application. However, large amounts of data associated with each object may become cumbersome. Further, a single map for plural objects may also become awkward to handle.

Evaluation of object characteristics sent by object applications can often be time-consuming while not resulting in the selection of the best metadata map, and thus the most appropriate data may not be accessed. Further, depending upon the amount of data available in a data store and the number of metdata maps available, the selection process may become very slow and cumbersome.

However, there is a definite need for translating between a wide variety of different data formats, other than just the relational to object translation of the basic system of U.S. Pat. No. 5,857,197. There is also a need for transmitting quickly and easily between various types of databases. Further, there is also a need for editing or otherwise altering data exchanged between databases for the purpose of facilitating object applications. The basic system of U.S. Pat. No. 5,857,197 needs further support to facilitate expeditious operation, both for its basic system and for modifications to its system that will accommodate better security and for selectivity of data.

The enhanced operation and efficiency of the basic system by U.S. Pat. No. 5,857,197 is enhanced by the inventive modifications depicted in FIG. **7**. The goal and purpose of the modified operation of FIG. **7** is to more quickly handle greater amounts of data by streamlining or otherwise enhancing the operation depicted in FIG. **1**. The operation of FIG. **7** also facilitates greater flexibility in that the system more easily handles multiple maps for a single object as well as use of plural objects for a single map. The reconversion process of FIG. **7** also allows conversion between database formats of a wide variety. As a result, editing of data for a particular object application (or other type of application) is more easily facilitated. Further, changes in data for a particular object need be done only once if such changes are handled in accordance with the invention depicted in FIG. **7**. A key attribute of the process of FIG. **7** is the enhanced processing speed facilitated thereby.

In the system of FIG. **7**, steps **1**(*a*) and **2**(*a*) are carried out in the same manner as in the process of FIG. **1**. It should be noted that the invention of FIG. **7** does not require that only objects be used to obtain data from relational databases. Rather, virtually any type of database can be accessed by virtually any type of application. However, for purposes of presenting a coherent example, translation relational database to objects required by an application will continue to be used. Also, as with the FIG. **1** embodiment, an abstract or translation layer is used to receive the request for data related to the objects provided by the object application. It should be noted that the abstract layer, which carries out much of translation activity can be constituted in a number of different ways. For example, the abstract layer can be the computers and communication links of a third entity (in addition to the application entity and data manager entity). One example would be the business entity Thought, Inc., the Assignee in the instant application. Thought, Inc. would provide all of the necessary software for the various translations and other operations that would provide the translation of relational data into object oriented form required by the object application in a manner that appears to be transparent.

However, the abstract layer need not be confined to a single business entity for a single group of computers. Rather, the functions of the abstract layer can be distributed over a number of different entities and associated computers, linked via the Internet, a number of different intranet arrangements, temporarily dedicated lines for fax or FTP transfers, satellite links, microwave links or any other communication link which will be used for exchanging data.

In another alternative, the abstract layer could be associated with the data manager, and be operated as a part thereof. However, depending upon the scope of data handled by the data manager, operations of the abstract layer could be limited thereby. In yet another alternative, the abstract layer would be part of the operating system of the entity operating the object-oriented application of software. However, the same limitations would apply thereto, as indicated for the situation with the abstract layer was part of the data manager. Clearly the most flexible arrangement is for a separate entity to handle the abstract layer, providing access to a wide variety of different customers operating these types of application software, and providing access to a wide variety of data stores and data store types. It should be noted that growth of such arrangements with many application user customers, accessing a wide variety of databases, leads to certain problems of scale. Many of these are addressed by the further system operation depicted in FIG. **7**.

Rather than going through the process of extracting object data for conventional evaluation at step **5**, this cumbersome and often slow operation can be bypassed by a cache check operation (step **51**). While the cache check at step **51** can be carried out in the abstract layer as indicated in FIG. **7**, the present invention is not limited thereby. Rather, the object data can be also be contained in a cache to be checked at step **3**(*a*) at the application site. In this manner, the necessary object data can be transmitted directly to the abstract layer and used for the evaluation with the metadata (step **6**) without going through the cumbersome extraction process.

It should be noted that the cache check operation **51** can be carried out by a plug-in software addition. In the alternative, the abstract layer may act as another database at a remote location. Apparently, the best method is to keep a cache memory located within the processing computers of the abstract layer to more quickly facilitate the cache check **51**. This allows the manager of the abstract layer to control the object data that is cached for easy re-use.

In a preferred embodiment, the present invention to provides a fully synchronized caching system that utilizes the transaction coordination facilities of a server such as a J2EE application server (for example, Weblogic 6.1), such system comprising a local or distributed computer system having a first data source referred to as the primary data source, a second data source referred to as the cache data source and the cache data source is associated with an object to relational mapping layer to provide a data source cache for object applications, and a server having a transaction coordinator with the ability to register multiple data sources, wherein:

(a) both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and

(b) the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and

(c) registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be sycronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

A number of primary object databases exist in the art. Non-limiting examples are: (a) Gemstone by Gemstone, Inc., Beaverton, Oreg., (b) Itasca by IBEX Corporation, Minneapolis, Minn., (c) Jasmine by Computer Associates International, Inc., Islandia, N.Y., (d) Matisse by ADB, Inc., Redwood Shores, Calif., (e) NeoAccess by NeoLogical Systems, Berkely, Calif., (f) O_2 by Ardent Software, Versailles, France (g) Objectivity/DB, Objectivity, Inc., Mountain View, Calif., (h) ObjectStore and Javelin, both by Object Design, Inc., Burlington, Mass., (i) ONTOS DB by ONTOS, Inc., Lowell, Mass., (j) POET by POET Software Corp., San Mateo, Calif., (k) UniSQL by Cincom Systems, Cincinnatti, Ohio, and (l) Versant by Versant Object Technology Corp, Menlo Park, Calif.

A number of primary relational databases exist in the art and have been described earlier in text that was presented above.

A number of SQL searchable memory resident relational databases that utilize JDBC drivers exist in the art, which may be adapted for use as a secondary database buffer in the context of the present invention. Non-limiting examples are: (a) HypersonicSQL, an open source relational database (publicly available for free at http://sourceforge.net including source code) that is simultaneously disk and memory resident, (b) TimesTen by TimesTen Performance Software, Mountain View, Calif., and (c) Polyhedra by Polyhedra, plc, Bellview, Wash.

A particularly preferred embodiment of the invention provides a cache data source can be set up as a memory-resident entire database, a disk resident database, or both a memory resident database and disk resident which are synchronized. In one object, of the invention the memory resident database may be a portion of the disk resident database and the size of the memory for the memory resident database can be set by the user.

One embodiment of the present invention provides a dynamic distributed or local caching system for an object model or one or more datasources by utilizing a call back API and at least one listener object that utilizes the call back API to listen for when changes to an object are to be pushed to a datasource or persisted. In a preferred embodiment the main listener object will broadcast such changes to any other registered listener(s) such as a navigation object, session bean persistence objects, or other such listeners. In a distributed caching system, a main session bean for persistence can be a listener for changes to any objects or object models that are registered to be persisted and may broadcast such changes to other session beans or object model navigator object instances.

In such an embodiment of dynamic distributed or local caching, when a listener detects via the call-back API that a change to an object or object model is pushed to a database, the object or model is marked by the listener as being dirty or invalidated, and the system can be set (parameters, flags, or property files) to provide different ways to handle this change in the cache. For example, an object or object model can be re-read from the datasource to obtain the changed object or object model and update the cache. In some situations, the listener can be set through the call back API to obtain data from the changed object and the broadcast notification of the change to registered listeners can contain the new object or object model values which can be used to update the local copy of the object or object model instead of obtaining the updated object or object model information directly from a data source. Also, a system can be set to notify the user that an object or object model cannot be changed because they have has been updated by someone else in the system, and to notify the user to simply obtain an updated copy of the object or object model or to mark their copy of the object as being old and not useable by the system.

Below is a general example explaining how a plug-in can be used as a database synchronization mechanism across multiple databases or multiple users such as across a first database and a second database. This example does not show (but the user can envision in view of the above description) where a listener class is checking for any push of changed objects or object models to a data source such as the caching data source or the main data source. The example below involves two data sources, but could easily have been two users of a cache, object, object model or data source. Two different instances by the same user are logically two users.

Example and Explanation for Using the CocoBase Plug-in Architecture to Synchronize Heterogeneous DataBases

Suppose the existence of 2 databases, B**1** and B**2**, with different schemas, S**1** and S**2**.

Assume both B**1** and B**2** have their own CocoBase repositories R**1** and R**2**, where M**1** is a map defined in R**1** and M**2** is a map defined in R**2** and both M**1** and M**2** are optionally identified by the same map name.

Suppose that the system uses CocoBase (commercially available object to relational mapping tool having a mapping repository capability) and a connection of CocoBase with B**1** is established as follows:

```
CocoDriverInterface connB1 =
CocoDriver.getCocoDriver (cbdriver, jdbcdriver, jdbcUrl +
";cocoplugin=thought.CocoBase.SynchPlugin", user, passwd);
"thought.CocoBase.SynchPlugin" is the name of the plug-in class
that is configured to replicate operations of database B1 to
database B2. For example, for the "insert" operation, the plugin
implementation would be similar to the following
(suppose replicationDB is set to connection to database B2):
    public int insert (int code, CocoUser userObj, Object
keyObject, String objectName, Integer objCount) {
        switch (code) {
        case CocoServerPlugin.CB_PRE:
            break;
        case CocoServerPlugin.CB_POST:
            Object o = null;
                if (objCount.intValue ( ) >= 1) { // if object
was insert in B1
                    this.replicationDB.insert (keyObject,
objectName); // we insert it in B2
                }
            }
            break;
    default:
        throw new RuntimeException("Invalid plugin!");
    }
    return CocoServerPlugin.CB_CONTINUE;
}
```

Once the above connections of CocoBase are established, any accesses to the database B**1**, such as:

```
connB1.insert (. . .)
```

will have the plug-in access B**2** and replicate the operation that was conducted on database B**1**.

The same mechanism described above can be used to replicate the result of other database operations (e.g. update, select, delete, etc.) that are performed against B**1**. Further, the application that uses CocoBase maps to access B**1** and B**2** can have CocoBase set B**1** as the primary database and B**2** as the cache database. In such a situation only the cache database B**2** needs to be accessed to retrieve data when data that the application needs is present in B**2**, but both data sources need to be accessed in order to insert or store data. This can significantly speed up data retrieve by avoiding disk accesses for data already present in a memory cache.

In the case where B**1** and B**2** are both registered as data sources in an XML descriptor file with a J2EE application server (for example, Weblogic 6.1) in combination with the CocoBase plug-in cache facility, a fully synchronized caching system can be provided that utilizes the transaction coordination facilities of the server. Very important and significant database caching abilities along with transparent data persistence is provided while providing transaction coordination that is controlled by the J2EE application server.

To more specifically describing the cache check **51**, in virtually any situation involving the cache check **51** of CocoBase, separate transactions will be carried out for cache check operation. In such a transaction a check of the cache will be made. If found, the relevant object data will then be moved to the next step of the operation for evaluation with the metadata (step **7**). If, on the other hand, the relevant object data is not found, the evaluation process takes place. This is controlled by the abstract layer manager for purposes of the present example, and includes an evaluation of the object data to determine what should be placed permanently in the cache memory that is used by the abstract layer for containing object layer data. Such an evaluation can be carried out for adding object data to a cache, whether or not the cache is controlled by the manager of the abstract layer. For example, it is entirely possible for data manager to carry out such a caching operation more easily supplying those metadata maps that are requested most often.

In the parallel activity occurring at the data manager, steps **1**(*b*), **2**(*b*) and **3**(*b*) are carried out in the same manner as the operation of the system in FIG. **1**. The difference is that at step **4**(*b*) selected metadata maps are converted to the XML format. This is a format already known for its facility in the handling text, thereby making modifications to the text of the map relatively easy. Selected metadata map or maps are loaded at step **6** into the abstract layer. The evaluation process at step **7** takes place to determine how to construct the actual commands to access the necessary data from the data store in order to satisfy the request from the object application (step **8**) in XML format.

The use of XML format facilitates modification of the map with simple text editors. Once the map is altered (to determine how data is to be stored and arranged), the application user can then use the map to effect the desired data handling for any associated objects. As a result, the data is easily manipulated without extensive recoating or programming expertise for the data store architecture, as is required with conventional systems.

Another alternative that speeds the overall operation of the translation process is to convert selected cached objects into XML format at step **52**. In effect, the system is carrying out objects to XML mapping to facilitate faster operation to the overall system, as well as easier modification of the object and map.

Based upon the presumption of a future single (or several major) XML standards, it would be practical to export not only XML map definition but also the production data accessed by the map definitions. Once formed, the data can be easily distributed to an entire network. Thus, step **8** of the FIG. **7** embodiments could be further modified so that the data is sent in XML format rather than just the map. XML is a highly desirable data exchange format since the data can be e-mailed to XML format more easily. The speed of translation and evaluation discussed for the FIG. **7** invention enhances the dynamic operation of the overall system in that maps can be loaded very quickly, responsive to requests based upon object data. Further, the map can also be altered dynamically (during the run time of the application) to suit the requester managing the application software. Further, the object to XML mapping would permit importation of data from XML format to any business object. Likewise, data from any business implementation object could be converted and exported in XML format along with its relationship, as defined by mapping tool such as metadata map.

Greater flexibility and faster dynamic mapping can also be achieved by taking XML data sets and using them to substantiate selected objects. An example of this would be to take a first XML document and utilize the information (data) within the first XML document to seamlessly substantiate objects in a second XML document. While this can be done by altering step **4**(*b*) in FIG. **7** to convert data store to XML in response to a provision of selected objects.

Accordingly, there will be a substantial advantage in generating XML data documents for exchange between parties to carry out commercial transactions. Conversion to XML will also facilitate the use of data sources or data architecture other than relational databases. Examples of such mapping includes object-to-object mapping and JAVA to object mapping. Once algorithms translating between XML and object formats are standardized, translations between any type of system can take place seamlessly and very quickly. Mapping can also include the use of JAVA Server Page (JSP) format.

Once editing has took place either for the metadata maps or even the production data from the datastore, changes can be recorded in the operation of Asnapshots@ so that a record of the map or other data at various times can be studied. As a result, data management paradigms can be developed. Normally such Asnapshots@ are made at the location of the party requesting the data (such as the object application user) once map modification is completed. The modified filed can then be placed in a source code management system.

The snapshots or map modifications, or even general data of modification is facilitated through the use of caching. Since a variety of different types of data (including metadata maps, objects, production data, and the like) are used as options in the inventive system, a number of caching methods must be adopted in order to accommodate the various types of data. Examples of caches include: a write-through cache without transactional integration; a two phase commit integrated cache; and, a temporary working storage cache, which acts as a second database. All three of these tech-

niques are discussed in more detail below. The present invention is also used to distribute caching or database using a plug-in Java database. Any number of different caching arrangements can be applied to the system of the present invention, but are limited only by what is possible in the art of data storage. The various caching arrangements can also be used to facilitate global changes in maps and data.

If caching arrangements are to be used with CocoBase, the accessing and updating of information must be defined in some clear way to control the caching and avoid unclear or unacceptable results. Discussed below are three methods for our caching controlling, followed by two examples of ways we cache in memory as either a "database" in memory with a third party plug-in or a "caching in memory of data" where both are connected to the O/R run-time layer. When a user connects to the O/R layer and subsequent database they register any optional plug-ins for caching or extending behavior and this then controls the access to data per one of the three mechanisms. Such options can be very powerful when combined with transaction capabilities of some Java application servers.

Write-through Cache without Transactional Integration

The first type of cache design is a "write-through cache without transactional integration" (no two-phase commit). This model assumes that applications not going through CocoBase may also be updating data in the first database (data source). In this method if a store data attempt fails at the first database, then the corresponding data of the second database (cached data) is deleted and may be (or may not be) refreshed from the first database depending upon developer policy. If the store data succeeds at the first database, then the corresponding data is updated in the second database (cached data). Notification of storage attempt results can be sent to the user to indicate a successful or failed attempt to store data. Therefore, multiple users can be utilizing the same cache in this instance and multiple applicationw (e.g, applications that do not go through CocoBase) may be accessing the first database and data integrity can be maintained.

Two-Phase Commit Cache

The second type of cache design is a two-phase commit cache design. This type of caching only allows updates of data by applications that work through CocoBase, through an application(s) that is either a compatibly integrated application(s) working concurrently with CocoBase, or uses the same policies of updating as CocoBase. In this cache model, data is only updated if CocoBase, or the integrated application can update the data in both the first database (source) and the second database (cached data). A commit to change data in both the first and second database (or information cache) is obtained as a first step and store data (update) completion is done in a second step. Again multiple users can use the same database source (or multiple application if they are integrated properly with CocoBase) and data integrity can be maintained.

Temporary Working Cache

The third type of cache design is a temporary working cache design. This type of caching only allows updates of data by applications that work through this temporary working storage cache (i.e., memory buffer or a second database which may be completely or partially memory resident database). This temporary working storage cache is designed to shield a first database from access and act as the datasource to users. The updating and accessing of the first database can be defined by rules of the developer as they desire. This cache design assumes that database accesses by database information users are directed to the cached information through CocoBase or through another application. The information in this working storage cache is used to intermittently update or to not update the first database (data source) as defined by the developer. In such a model, transactional coordinated database updates and persistence should be synchronized by registering both databases with the application having transaction coordination, such as the J2EE server Weblogic 6.1, for example. In such a case, this cache design may be described as a synchronized transactional database caching model.

The CocoBase software suite provides distributed caching of a database by using a plug-in third party free java database (for example HYPERSONIC SQL), however any third party java accessible database could be used. For example, a CocoBase application user can define a map of a first relational or object database to be accessed, and then import the data into a memory cache through a plug-in java accessible database (second database). This can allows CocoBase or compatible integrated applications to control access to data as defined by a developer, and database users may re-directed to access the second database in memory instead of accessing the first database in order to speed up performance. This also permits a database to be distributed to multiple users or to multiple applications on a network, where CocoBase or pre-defined developer rules control the updating and persisting of data by storing back to the original database any changes made to the memory cached java accessible database according to one of the three caching methods described above.

A distributed cache may use optional mapping server plug-ins and control access to multiple server databases. In plain English, the data source object database such as Oracle is opened as a first server database, and a second server database (for example HYPERSONIC SEQUEL) is also opened as a third party memory or virtual memory resident object database. The second server database can be loaded into a designated mapping server memory area. Once the first database is accessed, the data is sent to the second server database for storage and also sent to the user. Future accesses to that same data are directed by CocoBase to the second database. Also, updates of data are directed to both the first and second database by CocoBase and/or by a transactional coordinator of a J2EE server such as Weblogic 6.1 where both databases are registered with the J2EE server.

Alternatively, some of the latest alternatives for caching use the caching capability of a server to which CocoBase is providing maps and connectivity with a database, or the caching capability of another accessible caching facility. Any stand alone caching facility that a developer connects with the O/R layer as describe above in the caching methods descriptions can be implemented as a cache for database accesses).

A caching facility may be utilized by CocoBase to create a single or multiple objects in accessible caching facility memory, which object (or objects) may be stored in a relational database (or really any other piece of data, data object or data stream) object database, etc. for possible re-use or multiple use by application(s). CocoBase, integrated applications, or pre-defined developer rules will then control access to data and can permit the developer to require a user to directly access the data in cached memory rather than accessing the original data source. This can really speed up performance, e.g., up to 40 times faster with the

combination of Oracle as the first database and Hypersonic as the second (cache) database than by using ordinary Oracle database accessing.

For optimizing performance of applications that utilize the CocoBase runtime libraries and other repositories, it is important to cache not only data, but also to cache the map (or maps) to be used in the CocoBase mapping runtime. The underlying mapping connection automatically does this in CocoBase. So for example if a user is accessing an instance of 'Customer', the underlying thought.cocobase.cocopowder software module will cache the map information for the customer map within the connection object.

In terms of CocoBase, map caching is when the software suite reads and caches the map created with a mapping administration tool such as cocoadmin into RAM (random access memory) of the computer system. Once this map is cached, the CocoBase runtime repository libraries can generate and execute SQL much faster, but the first step of reading a map into RAM can be relatively expensive resource-wise. The cached maps are kept internally by CocoBase in a pre-compiled form that is exceedingly efficient for the runtime to access. Maps by default are cached within a connection once they are loaded and used for the first time. It is also possible to cache maps for all connections within a Java Virtual Machine (JVM) by using the CocoBase URL extensions for controlling cross connection caching.

There are two mechanisms that can be used currently with the CocoBase enterprise runtime libraries to control map caching.

The first is to use a cache configuration file that specifies which maps to pre-fetch and to place in shared cache. This type of map caching is controlled through a syntax such as:

```
JDBCURL: cocorep=configresource:/resourcefile.propertieS
```

or for example

```
JDBCURL:cocorep=configfile:/opt/file.properties
```

The cocorep syntax can begin with a colon ':' or semi-colon ';' syntax, so it can be compatible with different computer systems. The resource syntax describes what config resource should be loaded from the CLASSPATH, either directly from the file system or from some archived file such as a JAR or Zip file. The syntax of the config file is that a list of MAPNAMES needs to be specified in a comma delimited form for the tag MAPNAME. A sample resourcefile.properties file might appear as follows:

```
# Map names to preload
MAPNAMES = Customer,com.mypkg.Address,com.altpkg.Order
```

Another type of CocoBase software suite map caching that can be done is based on an XML repository. A set of maps can be exported as an XML repository and either placed into an archived deployment file, such as a JAR or zip file, or the XML repository can be an un-archived file that is loaded directly from the file system. To specify an XML repository in the CocoBase URL, the syntax would look something like:

```
JDBCURL;cocorep=resource:/CocoRepository.xml
or
JDBCURL; cocorep=file:/mydirectory/CocoRepository.xml
```

Both the colon and semi-colon are supported as the separator character for CocoBase URL extensions. As a result ';cocorep . . . ' and ':cocorep . . . ' would be processed the same by the CocoBase runtime libraries. Some EJB descriptor syntax requirements place special meaning on the colon or semi-colon, so CocoBase makes it possible to use either of these two syntax choices.

Fortunately, map caching with CocoBase only needs to occur once a session or once a JVM depending on the option used, and then it is cached in RAM. This initial read and caching process should be taken into account if any performance tests are being created to test CocoBase. Using the per JVM map caching can be a great way to pre-load and pre-fetch maps so that performance numbers are accurate based on normal execution patterns.

The mapping server of CocoBase can be extended through custom developed classes called plug-ins. A CocoBase plug-in implements the Java interface thought.cocobase.cocoserverplugin. This is a standard Java interface, and any Java class, which implements it and provides a public default constructor can be loaded automatically by the server at startup. Examples of such plug-ins are the cache and connection pool that ship as source code in the demos directory of the CocoBase software tools suite.

A CocoBase mapping server plug-in will implement the following interface signature:

```
public interface CocoServerPlugin {
public int call (int code, CocoUser userObj, int objCount,
Object keyObject, String objectName, Vector returnObjs);
public int close (int code, CocoUser userObj);
public int commit (int code, CocoUser userObj);
public int connect (int code, CocoUser userObj);
public int delete (int code, CocoUser userObj, Object
keyObject, String objectName, Integer objCount);
public int getColumns (int code, CocoUser userObj, String
catalogName, Vector colInfo);
public int getValue (int code, CocoUser userObj, Object key,
Object value);
public int insert (int code, CocoUser userObj, Object
keyObject, String objectName, Integer objCount);
public int rollback (int code, CocoUser userObj);
public int select (int code, CocoUser userObj, int
selectType, int selectCount, Object keyObject, String objectName,
Vector returnObjs);
public void setCommObject (Object comObject);
public int setValue (int code, CocoUser userObj, Object key,
Object value);
public int update (int code, CocoUser userObj, Object
keyObject, Object newObject, String objectName,Integer objCount);
}
```

In such an implementation, each method will be called before and after the associated operation (such as select) and the code will specify whether the action is a pre or post process operation also defined as public static codes in the plug-in interface API class.

```
public static int CB_PRE=0;
public static int CB_POST=1;
```

The Post condition will always be called unless the plug-in aborts or finishes the operation in the pre-condition

of the plug-in. Otherwise the plug-in will finish its pre processing, continue into the actual mapping connection, and then call the plug-in again with the post processing flag. After the completion of the post-processing, the resulting Objects or return codes will be passed back to the client process.

If the plug-in wishes to intercept and 'finish' an operation in the pre-process state without actually having the Mapping layer become involved, then the CB_FINISH return code can be returned as the return code of the method. If there is an error in the pre-processing and it is desirable to abort the request once again without involving the actual mapping layer, then we can return a CB_ABORT code. Otherwise the CB_CONTINUE should be returned (an all is well code).

```
public static int CB_CONTINUE=0;
public static int CB_ABORT=1;
public static int CB_FINISH=2;
```

Since all select operations come through the single 'select' callback, a software application may need to know which method the client called. CocoBase uses a code to distinguish what the client actually requested the call with the following values. This tells CocoBase if a select, selectBlock, selectNext or selectAll was called by the client, and the software can respond accordingly.

```
public static int CB_SELECTONE=0;
public static int CB_SELECTBLOCK=1;
public static int CB_SELECTNEXT=2;
public static int CB_SELECTALL=3;
```

Since a callback may be shared information, CocoBase needs a way to determine the 'context' of which user and which connection the callback is dealing with. In each of the methods, an instance of the thought.cocobase.cocouser is passed into the plug-in. This is the context object, which contains all of the information of the user, its underlying mapping connection and any other private data that the plug-in wishes to store in the object. Each of the cocouser instances has the following public 'get' interfaces (note that each has a corresponding set operation available):

```
public class CocoUser {
   public String getCocoConnection( );
   public String getCocoDriver( );
   public String getCocoURL( ) {
   public Object getDBConnection( ) {
   public String getPassword( ) {
   public String getUser( ) {
   public Object getUserInfo( ) {
}
```

The getUserInfo( ) method (and corresponding setUserInfo( ) method) is used by CocoBase to retrieve a plug-specific object. This can be any object (include lists of other objects) that will prove useful in the plug-in. This provides a very simple and flexible system that is fully extensible, and still easy to program.

There are several examples of plug-ins included with the published CocoBase evaluation distribution, and analyzing the working programming source code included as part of the distribution is often the best teaching mechanism. The caching mechanism makes extensive use of many of the features described above.

Client Programming Applications can Access Multiple Server Databases at Once

The mapping server provides a convenient platform to access and map and enterprise set of data sources, not just a single database. This provides a convenient system for porting data from one data source to another data source (see above).

Below are several examples of how a 3-tier client using CocoBase can access multiple databases from a single client object programming application:

1) The client can open up multiple connections to the same (or different) CocoBase Server—each having it's own database parameters and logins.
2) The server connection can access multiple physical databases through a software product called a 'Gateway'. These systems create a virtual relational database over top of a user's physical databases. Vendors such as Computer Associates, Oracle and B2Systems all have Gateway products that can be accessed from JDBC drivers, and such JDBC drivers can easily be connected to CocoBase.
3) If both sources being accessed are Oracle, Oracle DBLinks can be used to link in tables from other instances into the current instance being accessed.
4) Multiple physical connections can be accessed by using a plug-in which is easily obtained by extending one of the sample plug-ins shipped with the CocoBase evaluation distribution along the lines described above and in the instructions of the distribution, and loading the extended plug-in into the Mapping server.

A simple example for extending one of the shipped plug-in would be to have two open connections, one connection to a first database and a one connection to a second reference database. When the client wishes to first try to obtain data from the second database (for example, the second database is a faster or smaller memory resident database), a shared connection to the second database is left open in the plug-in, which would append a list of objects available from that database to the getcolumns lookup. The connection to the second database would integrate into the selected plug-in to provide a pre-process state so that data requested from the first database by the client program will cause the CocoBase runtime libraries to first look in the second reference database for an object that is present in the reference list of available objects before attempting to access the first database that also has an open connection.

There are other ways for modifying the plug-in to accomplish the same result. For example, a plug-in could be modified to use the cocouser storage area to store additional connections or lists of connections to be accessed first in response to efforts by an application client to look up objects.

Object Caching for Performance Enhancement

Object caching is provided by reading an Object (usually representing an underlying database row or fetched result set) instance and placing the data in memory for future reference while causing the database connection of CocoBase to remembers that the data is now in memory for future reference and it is unnecessary to access the database for that data. Using this technique, if the same Object instance is requested more than once by a programming application, it can be simply pulled from memory, instead of having to be retrieved from the physical database and the related over-

head associated from that operation. Since subsequent reads are read from the in-memory copy instead of the underlying data storage system, this technique can be VERY efficient and highly enhance performance. If the database server is over burdened, this can become essential for optimizing and increasing overall system performance. Since the Object Cache allows users to retrieve data quickly and efficiently from memory, it can often avoid the need for object look-up again, even with multiple clients.

With CocoBase Object Caching, as the data for each object (or the object in the case of accessing an object database) is loaded from the data source, the loaded object is stored in an in memory object cache. The Object must be uniquely identifiable to be cached—in other words a 'unique' primary key field or composite key fields must be defined. This cache is shared by all of the clients accessing the Mapping Server, and, as a result, only one copy needs to be in memory at once.

The CocoBase Object Cache contains a method for looking up the object in terms of its unique object identifier column(s). Before an object is loaded from the database by the CocoBase runtime repository, it is looked up in the cache by the identifier column(s). Using this technique, an object present in the cache is loaded from the cache instead of requiring the system to access the data source and create a new instance of the object. If the object is found in the cache, then that object is returned from the cache. A developer could create the cache in a way that would cause it to grow and fill all available memory, so it is wise to use a large memory size for the Java startup parameters. If a VERY large set of tables needs to be cached, then something like the Gemstone/J Java virtual machine should be used. Their virtual machine can grow with its own virtual memory, allowing the cache to grow as needed.

Caching with the CocoBase Enterprise Optional Mapping Server

CocoBase Enterprise provides two separate examples of a shared central cache source code examples of optional mapping server plug-ins. These programs are cocoserversamplecache.java and cocoserversamplecachepool.java, and each is included as both source and pre-compile binary forms. The plug-ins can be modified for custom caching, or they can used as they are shipped to provide useful out-of-the-box shared caching. The shipped compiled binary code classes belong to the thought.cocobase package by default, and should be copied into the subdirectory . . . \classes\thought\cocobase if the developer finds it necessary to modify the source code and then re-compile the modified code to provide new pre-compiled binary forms.

In the demos directory of the CocoBase package there is provided a cache configuration file, which can be used to configure Map keys that will be used by the system for caching features. The file cocokeybundle.properties found in the demos directory of the installed CocoBase software package, can be edited to customize caching easily. Just add entries for each map which describe the key or keys of that map, which will comprise a unique 'key' utilized in identifying each unique instance. The format is 'Map=COL[, COL2[,.] . .]'. To provide an index for the Map 'customer' using both attributes 'name' and 'address' as the composite unique key, place an entry in the cocokeybundle.properties file which has the following format:

customer=NAME,ADDRESS

Note in the above entry that the attributes are ALWAYS upper case, and that they match the 'virtual' column name used in the map (for example :NAME and :ADDRESS even if columns are NAM and ADDR).

The client (presuming they have permission through proper assignment of their admin ids by system administrators) can "additional override" this default value, but using a cocokeybundle.properties file will allow the caching to be configured on the server without having to code or make client specific changes. By using this default properties file, each column that is included in the properties file is made part of the lookup key by the cache.

In some cases, it may be desirable to allow the client to configure a cache, and the following example code accomplishes the same task. This example code is extracted from the demo program cocodemocache.java included in the demos directory of the installed CocoBase software package. The example code uses the CBSETCACHEKEY "virtual Object" and "procedure call", both of which can be intercepted in the caching plug-in, and the plug-in can be set to over-ride each of them to manage caching configuration.

```
//
// Set up a searchable cache key for the Customer Map.
// Since the demo database doesn't have primary keys supported
// in JDBC, we can simulate a primary key by configuring.
//
Vector cbkeyfields = myBase.getColumns("CBSETCACHEKEY");
  if(cbkeyfields != null) {
    System.out.println("Server is cache Configurable,
configuring Customer object cache.");
    // Configure Key information for Customer so server caches
it
    // properly. We only have to do this because
    // SimpleText doesn't support primary Key fields. If it
    // did, this wouldn't be
    // necessary.
    GenPObject gpo = new GenPObject( );
    Properties props = new Properties( );
    // We want to index on the NAME attribute
    props.put("KEYLIST", "NAME,ADDRESS");
    props.put("OBJNAME","Customer");
    props.put("CBADMINUSER", "cbadmin");
    props.put("CBADMINPASS", "cbadmin");
    gpo.setPropObjectData(props);
    myBase.call(gpo, "CBSETCACHEKEY", 0);
  }
  else
    System.out.println("Server either doesn't support Caching
or isn't cache configurable.");
```

While it is much simpler to configure caching on the server through the properties file (and preferable) as described above, client side configuration is also possible using the technique described above. Both plug-ins keep the cached objects in a local in memory server cache until the changes have been committed into the database. Once the changes to objects are committed into the database, the cached objects are merged into the central shared cache and are then accessible by other clients needing those objects.

Performance can be enhanced by maintaining a list of previously instantiated objects in an "in memory" cache list of objects in the Optional Mapping Server. This can be readily accomplished through using a variety of data structures and facilities including hashtables, vectors and object databases such as Gemstone (which use a technique called "persistence by reachability") to extend the Java Virtual machine to have Virtual Memory.

System Performance Increases are Possible with Caching and CocoBase

CocoBase can support several thousand direct database requests per minute even on a simple Pentium server with garden-variety databases. Caching provides an even better scaling promise, especially for heavily accessed databases. Services such as Caching and connection pool management can significantly enhance performance numbers. Tests of the caching methods when CocoBase was being improved to implement the above concepts have shown that performance ca nearly triple through the use of even simple caching techniques. For example, some internal benchmarks with CocoBase and caching have clocked a 233 MHz K6/Pentium processor machine with Windows NT and 64 MB RAM as being able to serve up about 20,000 database requests per minute for individually requested cached data objects. The 20,000 requests did not constitute high volumes of data (about 30 bytes per request—a simple Object), but nonetheless such benchmarks can give developers a "rule of thumb" idea of potential increases in throughput due to caching, even when a very low-end and non-tuned system is used to serve up responses to database requests.

Other tests with CocoBase and caching used a small matrix test suite to show approximately how many cached read operations per minute a developer might be able to expect on a sample Platform. The numbers also indicated how accessing blocks of data can actually increase the overall network throughput of objects that a server is capable of handling. For example, blocking data in access blocks of 100 instead of access blocks of one can actually increase the server throughput to 5 times the data volume even when though the number of actual requests, which could be served, was reduced. These tests might encourage developers to use the block API wherever possible, and then tuning the access block sizes to an appropriate size for an individual application to enhance performance.

In some caching tests, a block size of 20 had a higher throughput overall than a block size of 100, a fact that may be related to the mechanics of how CORBA handles large blocks of I/O.

The above performance numbers might be merely considered guidelines, since performance numbers such as this can be very network, ORB and data specific. However, these test do clearly demonstrate that using CocoBase with caching (Mapping Server Cache) in combination with the tuning of block size can have an significant impact on performance, and can sometimes provide a significant throughput increase for application/database servers.

There are obviously additional ways to tune specific environments and configurations even outside of CocoBase, since the above numbers are mainly given as approximations that demonstrate the CocoBase system and technology that can give developers real world examples of tuning performance, and perhaps a ballpark idea of what might be possible. Performance enhancement of an individual system and application configuration may vary based on Java Runtime, OS, Database(s) used, networking configuration including and bandwidth capacity, ORB, etc.

Where caching really can really be beneficial to a system is in the client's 'perceived' performance, and when 'blocks' of data can be successfully cached (i.e., the application does not need to wait for an SQL statement to execute in order to access and use data for processing a request, and the system does not have to wait for the same unchanged records to be accumulated in memory over and over again), a system can seem faster because of the reduced processing delays. The CocoBase system uses caches that are written to serve multiple users out of a single pool, and by default the CocoBase cache system is a write-through cache for extremely scalable performance numbers.

Such a write through cache is written for a CocoBase system designed to provide scalable performance while avoiding certain undesirable impacts on the transactional capabilities of individual application servers. Since caching is a complex science, it should not be tackled at the beginning of a project that uses the CocoBase system. Instead caching should be implemented and fine-tuned as a final step in development. Such techniques can allow a developer to refine and improve an already streamlined system and avoids using a cache during development that could lead to unfortunate masking of poor software designs and system design flaws. Such problems might only become evident when the final application is implemented and scaled, if performance issues arise, which were masked by the caching during development stages.

Enabling Optional Mapping Server Object Caching

In the Optional Mapping Server that can be used for object caching, caching is implemented in the Server Plug-ins instead of being within the Optional Mapping Server itself. This can provide more flexible caching that is user extensible. If either the cocoserversamplecache or cocoserversamplecachepool plug-in is loaded, all objects having a database primary key that is accessible from a JDBC will be cached by the CocoBase system. In this system, object data caching is done through the Optional Mapping Server plug-in APIs. Source code jump-starts for providing caching are included with the Optional Mapping Server as source code, but the CocoBase system does not automatically integrate caching directly into the Optional Mapping Server. Such integrations should be saved until the later stages of project development to avoid masking design flaws.

At the final stages of development, a developer can utilize the source code provided with the CocoBase package and implement a specific kind of caching necessary for the design of the developed system, and integrate that caching system into the Mapping Server. As described above, when the source code of the examples provided don't provide explicit functionality desired by a developer, the code is extensible and can be modified to meet specific project requirements.

Sample cache provided can yield many of the common features needed by developers when the CocoBase system is implemented, and the sample cache will probably be completely useful without modification for many tasks. Modification and implementation of caches are an advanced developer functionality issue. The steps of implementing and tuning caching should only be taken on as a final step before deployment, and be taken on with careful planning. Blindly implementing caching without fully understanding a system design and understanding the purposes for caching in the context of that system design can create serious performance, memory and data integrity issues. For example, having multiple databases open in combination with caching can raise real issues with respect to data integrity if caching and multiple databases are not managed properly, particularly where database accesses are not properly managed and synchronized.

Managing Object Integrity for Applications Using a CocoBase System

Object integrity is typically maintained by keeping the cache synchronized with the data in the underlying data store, and by ensuring that any changes are reflected in the cache if, and ONLY if, such changes are successfully

committed to the database. When a programming application ends a transaction with a commit operation, any updated persistent objects are written back to the data store, locks are released, and the client specific cache is invalidated (this is referred to as integrating the client specific cache into the shared cache). When the application ends a transaction with a rollback operation, updates are discarded, locks are released, and the client specific cache is invalidated. This will leave the shared cache unaffected and unmodified.

For some application environments, objects may additionally need to be retained in the cache across transaction boundaries (in other words it may be desirable to integrate a change into the shared cache before it is committed). In this case, some application-specific logic is needed to deal with the need for consistency between the objects and the underlying data store.

Another important issue associated with Object Integrity is the need for unique Object Instances. If duplicates are allowed, it can become nearly impossible for a system to cache and to track Object Instances without the use of the transactional capabilities of a server to synchronize the cache and the main datasource. Because of this, some 'composite' key may be required to ensure that unique instances exist. If a unique key definition exists, then the Mapping Server can manage Object Integrity more properly.

Differences Between Map Caching and Data Object Caching

Object Caching and Map Caching should not be confused. Object Caching caches the user data, and must be implemented in a Optional Mapping Server plug-in. Map Caching automatically occurs in a CocoBase connection. Since Maps can become quite complex depending on the number of where conditions and fields involved in its definition, it becomes critical to Cache connections by default. As a default, CocoBase meta data for each Map is cached by the CocoBase system the first time that the Object is accessed.

The meta data caching can be triggered for a particular map before the data from the map is utilized by calling the method getcolumns( ) on the Connection with the Map name as the parameter such as:

```
Vector tempVect = myBase.getColumns ("Customer");
```

In this example, a customer Map will be accessed and cached in the CocoBase runtime before any SQL is created or allowed to execute. This is a convenient method designed to speed runtime access of Maps in the Mapping Server through pre-loading and caching the Map. If this pre-loading and caching is necessary for Maps that are likely to be accessed by clients, it can be done at server/connection startup in order to improve perceived user performance characteristics. Once the server/connection is achieved the system is ready for database accesses, there is nos second delay required to pre-load and to cache the maps before accessing data.

Because some systems may have literally thousands of Maps for a single connection, the CocoBase runtime is not configured by default to auto-load all maps, only those specifically requested to be loaded or accessed directly will be automatically loaded. In the case of Map caching, a lazy cache is more effective. Especially since with a lazy cache it is so easy to trigger a load and cache of particular Maps that are likely to be used.

CocoBase is currently configured to only permit an application to dump a Map cache from a connection by closing and re-open a connection. Such a dump involves only two method calls, and is easy to implement if necessary by simply entering calls similar to the calls to the "MyBase" database that following two lines illustrate:

```
MyBase.close( );
MyBase.connect( );
```

These two lines will cause all of the Meta Data Map information related to this database to be reset from the connection.

Object Uniqueness Requirements in a Cache

Since it is possible with most databases to have duplicate records in a table that maps the data structure, this can prove problematic for providing a caching function that can server up unique Object Instances. The CocoBase system and runtime resolves this issue by using a default setting with respect 'unique' keys. This default setting will only permit caching of those records that are specified by a 'unique' key, which identified through the Database Meta Data available from JDBC. Additionally, it is possible for a developer to define a unique key column (or set of columns) through a startup init file or through the runtime API. Only a developer with advanced system design understanding and high level programming skills should attempt to make such a modification, since this is a task that should only be done if instances to be stored in the cache through the map are truly unique.

The CocoBase system is designed to manage the cache by using a unique key, or the key column(s), as the unique identifier in looking up instances and then stores the instances in a Java Properties Object with the unique key as the identifier. With this design, if two duplicate records are retrieved from the database, only one instance will be preserved in the Properties based cache. The CocoBase is not designed to create uniqueness or to enforce uniqueness for database tables that do not have unique index defined. Because of this, the CocoBase system is designed to ignore an attempt to create caching or to use caching if a unique primary key has not been defined in the database, or a unique key is over-ridden by the developer by their software application design.

The CocoBase system is designed to allow non-unique objects to be accessed just as in ordinary database accesses where the CocoBase system is not used. However, the use of as caching is not turned on for those Objects. If a database table doesn't define a unique key, the Objects simply get passed through the CocoBase plug-in without caching becoming involved. However, a developer can easily take advantage of CocoBase plug-in caching even if a database table does not use a 'unique' key column set, by simply defining their own unique key for the database table.

Developers can look at the cocokeybundle.properties startup init file to see how the CocoBase system provides an ability to dynamically define a Unique Key. Also, developers can review the cocodemocache.java file portions related to the CocoBase connect( ) methods format and commands, for an example of how a developer can easily define an Object Unique key through the use of the Mapping server CBSETCACHEKEY functionality.

Design of the CocoBase Sample Optional Mapping Server Object Caching

As described above in several text locations, there are two sample Caching plug-ins included in the CocoBase system

distribution and the design of each plug-in is bid different. One plug-in is a "Cache only" plug-in with the file name cocoserversamplecache.java, but the other plug-in named cocoserversamplecache.java is designed to provide a combined caching and connection pooling.

The cache portion for each plug-in is basically identical to one another, and the cache design is fairly intelligent with respect to caching systems. The cache portion of the plug-in keeps a rollback log for each client that is using the cache (a private cache) which is integrated with the shared cache upon the completion of a successful commit that updates the datasource. If the cache setting of autocommit is used, the caching system of the plug-in is designed to automatically integrate the changes to the private cache into the shared cache after each operation is completed that changes the private cache.

The cocoserversamplecache plug-in is loaded as the default plug-in for the script startcache.{bat/sh}. The other plug-in is used by substituting the script startcachepool.{bat/sh} and cocoserversamplecachepool is loaded as the CocoBase system plug-in instead of the default cache plug-in.

The caching portion of the plug-in is also designed to support some administrator control functions for starting, stop, and managing the cache. Additional features may be added by modifying the source code that is provided as part of the CocoBase system. The cache plug-in source code described herein (and to be released after the filing of this document) supports (among other things) the administrator control functions of CBSTARTCACHE, CBSTOPCACHE and CBSETCACHEKEY, which can be used by a developer to set a caching system design that will start and stop the shared cache, as well providing the ability to override the primary key column(s) used at a resource to lookup Objects in the cache.

The cache plug-in source code does not need to be modified to manage transactional coordination or regulate cache memory size management. Cache system design is easily configured to solve such issues in a much more reliable fashion. Cache databases, for example, ODI and HYPERSONIC SQL each implement both manual and automatic mechanisms to trim the cache size on low memory availability. Other cache management features are provided by server monitoring capabilities, such as the multiple database synchronization capabilities of Weblogic 6.1, as described earlier. If the cache database and the main database are both registered with the Weblogic 6.1 server (or other application servers having similar monitoring capabilities), the transactional capabilities of the server can be harnessed to manage synchronization of the cache database with the primary database in coordination with the CocoBase system and fully implement the cache plug-in features of CocoBase for enhanced system performance.

Optional Mapping Server Definitions and CocoBase System Integration

The CocoBase Optional Mapping Server is designed to integrate into and interoperate with standard Application Servers whose users need to have database access. The CocoBase runtime and Optional Mapping Server can be used to provide persistence and mapping functionality to the overall system irrespective of the type of object programming application server that is chosen to be utilized for a particular computer system.

Generally speaking, application servers enable corporations to develop, deploy, and manage business-critical applications that frequently need to have transactional coordination capabilities. Object programming applications that run under such servers can delegate certain functions, such as the transactional coordination to the server.

Many sound reasons exist for developing applications that can run under such servers and can utilize the CocoBase database access features for development and deployment. A few of those reasons are high performance, rapid development, scalability, easy integration with existing enterprise systems, state management, load balancing, and table/row locking mechanisms. Such features can be the keys to good html and Jave Server Pages (JSP) creation that can interface well with a user and permit the user to access to data and to conduct transactions that implement important features of the server system and the CocoBase system that are invisible to the user. Examples are the server transactional coordination and the CocoBase runtime and caching systems. Such software application programs can be developed to delegate the functions of database access and data persistence to the CocoBase system, without the need to include database specific or persistence specific code in the application program source code. Moreover, CocoBase can automatically generate the server connectivity source code and persistence source code from the database maps.

CocoBase leverages the functionality of the Optional Mapping Server and features of application servers to provide additional advance mapping functions—typically the weakest link in Application Servers. CocoBase has become the pre-eminent mapping technology in Java, and provides superior mapping that can be readily utilized with virtually any mapping server as part of the overall system design. Application servers should be thought of as very complimentary technology to the CocoBase database mapping features, the CocoBase mapping server, CocoBase cache plug-in features, CocoBase runtime SQL generation facilities as well as other functionalities of the CocoBase software system product line.

Connection Pooling Systems with the Optional CocoBase Mapping Server

Connection pooling may be described as the concept of keeping database connections open and re-using them with new clients. Since it can be a resource intensive prospect to open a new database connection it is often advantageous to reuse existing database connections when possible, and to even use cached objects that are obtained from an earlier access of the database. A connection pool can be created for multiple clients. It keeps the connection to the database open for future requests. Creating a pool of JDBC connections gives clients ready access to connections that are already open. Connection pooling can improve performance by removing the overhead of opening and closing a new connection for each database operation or each time a new user tries to access the same database.

The connection management service of CocoBase can determine whether a database query can use a connection from the connection pool based on data source, user name, password, and the logged-on user account, etc. The connections in the pool can be shared among any users who can successfully match the login requirements. With this system design, each member client can get a connection from the pool for use, and then return it to the pool when finished (but the connection itself is not closed).

To configure the connection pooling of the CocoBase system, the developer can simply edit the Pool configuration file to set the desire connection pooling requirements (the file cocopoolconfig.properties). Both the cocoserversampleconnectionpool.java example plug-in and the cocoserversamplecachepool.java example plug-in are designed to uti-

lize this property file in order to configure the system Connection Pooling. The source code of each Connection Pooling plug-in can be readily customized to meet any project requirement not covered automatically in view of the discussion and description provided herein.

CocoBase System Optional Mapping Server Overview

CocoBase provides a solid infrastructure to allow easy user plug-ins for state management, load balancing, caching, connection pooling, etc. Because the focus of this server is on mapping, it can be referred to as an Optional Mapping Server. Such a mapping server it is designed to tie together different data base driver systems, and provide transparent and consistent use of those systems for an application developer. Another good use of such a system it to permit the easy transport of data from one data system to another, as described above in detail.

In one implementation, the CocoBase Optional Mapping Server is designed as an ORB Service loaded into the CORBA ORB. In such a case, each mapping server is fully multi-threaded and capable of supporting many simultaneous users. The CocoBase Optional Mapping Server also provides access to and transparent use of JDBC and either CORBA or RMI with 3-tier CocoBase adapters to the Mapping Server, without requiring application developers to understand or to program in either of those driver systems. The CocoBase System also provides applications with an added level of security that is not possible from Internet enabled JDBC drivers.

The CocoBase system runtime, data caching, object caching, and mapping features can be readily utilized by a developer to achieve a very scalable and high-performance data access and mapping system. This is accomplished by minimizing I/O between the client and server, in a fully multi-thread multi server architecture. The developer can permit client applications to decide when and where and how much data is sent between the client and server. This means the system does not need to burden down data pipes with irrelevant information or with lots of small independent network requests. Because less I/O is used, a system can be designed which utilize the same data bandwidth to support many more users with the CocoBase system implemented than it could have supported in the absence of CocoBase.

Optional Mapping Server Security with CocoBase System

Object programming applications can delegate database accesses to CocoBase runtime and mapping repositories to create increased security on computer systems. This can be particularly helpful when the system is connected to the internet or other open connections. Applications that interface with uses are more secure if they have been written to utilized the CocoBase system and to delegate the database accesses to CocoBase. In such a case, clients (system users and application software) access the optional mapping server through a CocoBase map. Each map explicitly specifies each table field and where conditions associated with available operations, and the client does not have access to the underlying database driver. This can be utilized to restrict the data that can be access by a particular client. The CocoBase System provides this additional security because database access is done through the list of maps available to a particular user, and the client or application software does not have direct access to the underlying database driver and therefore to data other than that available through the list of maps.

The CocoBase system is designed to provide this additional layer of remote security by providing a list of maps to which a particular user is granted access. Even this security feature can be easily extended by customizing the source code of the Optional Mapping Server Plug-in.

The CocoBase Optional Mapping Server model does not place SQL in client applications (which reduces rogue client capabilities). Instead, the runtime libraries of the CocoBase system as described at length earlier above. The CocoBase runtime libraries in combination with the pre-defined maps manage all SQL access to the data source through the Mapping Server and predefined Maps. An implication of this design means is that if a given CocoBase map, an operation within a map, or an attribute within an operation isn't published to the client, it does not exist from the perspective of the client.

By contrast, the CocoBase Relational 2-tier system sits directly on the JDBC driver instead of utilizing the mapping server and the runtime maps, and the 2-tier system does not possess the same security mechanism. In such a system, the CocoBase 2-tier runtime libraries permit a getconnection( ) call, which returns information necessary to access the system JDBC driver. As a result, it's not possible to provide additional security over JDBC if such a 2-tier system is employed. Many system that are not truly an "O/R" system in that they are not fully delegated the control of database assess have the same problem. In such a system, security will be limited to whatever security is provided by the JDBC driver and the associated database, such as database login protocols and the like. Such login protocols can be limited. Typically, if a database user has the right to can log into a database, then all of the data in that database is accessible to that user. This can require a system design of multiple databases where part of the data is duplicated, which can be hard to keep synchronized and uses additional storage resources.

In view of these additional security features due to the CocoBase system design, developer that need to provide access over the Internet should use a 3-tier CocoBase adapter and the associated mapping server. The CocoBase Relational 2-tier system and other similar systems may be sufficient mainly for Enterprise 2-tier Intranet application development. However, even in such systems it may be desirable to have a single database where certain data needs to be secure from access by some system users. The Optional Mapping Server provides applications an added level of security not possible from Internet or Intranet enabled JDBC drivers. Due to the security design flaws present with typical JDBC drivers, and the concept of permitting a JDBC driver is to pass arbitrary SQL from client applications to back-end server systems, it can be difficult or virtually impossible to secure JDBC systems for Internet or Intranet use without the use of the CocoBase 3-tier system of delegated database access.

The CocoBase system may also be described as having architecturally secure design that works on the 'publish paradigm' with the server managing and publishing all access to SQL on the database server. Client access is restricted to only those defined maps, which are published as being available from the Mapping Server. With this Map focused design, the CocoBase Enterprise 3-tier system (with its delegated database access features) allows companies to pre-define maps that can be executed by clients, instead of allowing clients to define or change the SQL statement that the server is capable of executing. Through adapting the CocoBase plug-in source code, developers can easily integrate/extend the components for providing additional security, and to delegate security and project specific logic to the CocoBase system.

Maps and objects related to those maps can be created in a number of ways, usually in response to application users requests. Accordingly, the system of the present invention will have to accommodate such numbers of objects and maps associated with those objects. Because of the flexibility of the present system, maps can be bonded to objects dynamically (during the application run time). The result will be that a single map could be appropriate for a plurality of objects. On the other hand, a single object may be bonded by a plurality of maps. Objects that can be used to access large volumes of data, the use of any particular map may initiate a data response so large that it cannot be easily managed. Accordingly, single object may have a number of different maps each configured to access only a particular portion of the data associated with the selected object. In this manner, maps can be used to select only that data which would be relevant to the map user. This selectivity will render a large volume of data associated with a particular object to be more manageable for access and use. Likewise, the distributed of selected maps to access only selected portions of the datastore associated with a particular object can be used in as a security device. In this manner, individuals are provided with maps having the capability of accessing only those portions of the database that are appropriate for the map user. Thus, the selection and distribution of maps for a particular object can serve as a security hierarchy with respect to the data to be made available.

The dynamic mapping of the present invention can be applied where a Java object provides translation by mapping such an object to first data source (either relational or object) can also by mapping such an object to an XML for a second format database. This arrangement allows greater control as to how the data steps are exchanged, filtered, and validated by exchanging data.

While a number of preferred embodiments of the present invention have been discussed by way of example, the present invention is not to be limited thereby. Rather, the present invention is to be construed as including any and all variations, modifications, permutations, adaptations and embodiments that would occur to one skilled in this art once having been taught the present invention.

We claim:

1. A local or distributed computer system comprising a user interface, a working memory storage area and at least one device for permanently storing information, wherein a portion of said memory of said system is loaded with at least one portion of a computer software program containing logic for accessing data in a first data source having a first particular data structure and having a particular corresponding structural schema and the logic for creating or maintaining all or a portion of such data in at least one second data source having the same or a different data structural schema, wherein:

(a) at least one of the data sources is an object data source store or is data encapsulated by data objects located within an object application program, wherein the data objects are organized by a discernable organizational structural schema or are capable of being organized by an organizational structural schema,

(b) the computer system comprises a mapping system that includes logic for accessing each schema corresponding to a different data source accessible to the mapping system, and the mapping system may further include logic for generating a structural schema for one or more data sources when a schema is absent or is unavailable to the mapping system, wherein repository maps are organized according to at least one navigation model or schema, or the mapping system provides an interface and software logic for organizing or reorganizing repository maps and data objects according to at least one navigation model or schema,

(c) the mapping system comprises software logic capable of correlating, of translating, or of both correlating and translating, all or a portion of the data located in each of the different data sources that is being created or is being maintained by the mapping system, and

(d) the mapping system comprises a mapping repository that includes metadata, or includes software logic for accessing metadata, wherein at least a portion of the metadata defines relationships between two or more data objects located in a data source or located in an application program capable of being run on the computer system, and wherein:

the one or more object programming applications capable of being run within the computer system are tailored to delegate both the accessing of a data source and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanisms in the application code.

2. A local or distributed computer system according to claim **1**, wherein the mapping repository further comprises software logic, or access to software logic, for resolving mismatches between the data schema for one or more data sources and the data schema for two or more data objects located within an object application program that is running within the computer system, or in an object application program that is capable of being run within the computer system.

3. A local or distributed computer system according to claim **2**, wherein multiple schema maps can exist for a single data object and multiple objects can use a single map that is located in the mapping system repository.

4. A local or distributed computer system according to claim **2**, wherein the mapping repository includes one or more items selected from the group consisting of maps of database schemas, object definitions, other programming definitions or parameters, object metadata, database settings, and complex relationship definitions.

5. A local or distributed computer system according to claim **4**, wherein the mapping repository includes one or more items selected from the group consisting of maps of database schemas, object definitions, other programming definitions or parameters, object metadata, database settings, and complex relationship definitions.

6. A local or distributed computer system according to claim **1**, wherein the object application program which contains two or more data objects wherein metadata accessible by the mapping repository defines relationships between the two or more data objects, may be run independently as an applet, as a stand alone application program, as an application program or service within an object program application server system, or may be capable of running in two or more of the these types of programs.

7. A local or distributed computer system according to claim **1**, wherein one of the data sources is a relational data source, is an object data source, or is an XML data source.

8. A local or distributed computer system according to claim **7**, wherein one of the data sources is an XML data source.

9. A local or distributed computer system according to claim **1**, wherein one or more of the data sources is located

in transient memory of the computer data system and can be persisted to a non-transient memory data source.

10. A computer system according to claim **1**, wherein the mapping system portion is designed to provide different maps for particular objects to different users and does not permit direct access of computer system users to JDBC drivers for any mapped data sources of the computer system, and wherein this mapping system provides varying levels of access to the mapped data sources for at least two different users of the same system, whereby a user only has access to a particular list of maps that are available to the security level of that user.

11. A computer system according to claim **1**, wherein the mapping system portion is designed to provide to a system user who is accessing, creating or updating maps, or accessing objects on a system to make data changes related to a particular object and to promulgate the changes to that object as either local or global changes on the computer system.

12. A computer system according to claim **11**, wherein the mapping system portion provides an interface permitting an authorized user to edit or create the tables, fields, or attributes of a data map for an object as a table format or XML file format without requiring the user to have extensive knowledge of a particular relational database as a source of the data, or extensive knowledge about how to directly access that relational database.

13. A local or distributed computer system comprising a user interface, a working memory storage area and at least one device for permanently storing information and a mapping system and the system having a portion of said memory loaded with at least one portion of a computer software program containing logic for accessing data in a first data source having a first particular data structure with a particular corresponding structural schema and the logic for creating or maintaining all or a portion of such data in at least one second data source having the same or a different data structural schema, wherein:

(i) at least one object programming application running within said system is tailored to delegate both the accessing of a data source and the generation of SQL strings to a runtime library repository, which repository can access the database directly or through a database driver, such as a JDBC driver, without the need to imbed specific database accessing mechanisms in the application code, and

(ii) the mapping system of said computer system provides an interface and features that permit a user to access, create, or update the metadata of a map as a dynamic computer system update, without requiring the user to either open a new connection to the data source or to restart an object application program that is running while the user is dynamically evaluating or changing metadata for a map, and wherein the metadata of a map that a user can dynamically evaluate or change includes a map description of data or relationships between data, and wherein such map description is at least one member selected from the group consisting of a data source relationship, a relationship between at least two objects of an object application, and both a data source relationship and a relationship between at least two objects of an object application.

14. A local or distributed computer system according to claim **1**, further comprising a fully synchronized caching system that synchronizes the caching of a delegated data source access management object to relational mapping layer and the transaction coordination facilities of an application server, and the computer system comprises (i) a first data source referred to as the primary data source, (ii) a second data source referred to as the cache data source that is associated with an object to relational mapping layer to provide a data source cache for object applications, and (iii) a server having a transaction coordinator with the ability to register multiple data sources, wherein: (a) both the primary data source and the cache data source are registered with the transaction coordinator facilities of the server, and (b) the cache data source acts as secondary data source to speed up data accesses for an object application and the cache data is plugged into the object to relational mapping layer, and (c) registration of the cache data source with the transaction monitor of the server provides the feature that any changes to the cache will automatically be synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources, and optionally, wherein the transaction monitor of the server calls the mapping system and in the call to the mapping system delegates accessing or updating of data in the data sources registered with the transaction monitor, such that the transaction monitor does not complete a transaction until it is notified by the mapping system that a data source has been updated by the mapping system in order for any changes to the cache to be automatically synchronized with the primary data source or record upon transaction completion, including commit or roll-back of changes for both data sources.

15. The computer system of claim **14**, wherein at least one cache data source can be set up as a memory-resident entire database, as a disk resident database, or as both a memory resident database and disk resident and in each case the cache data source is synchronized with the transaction monitor of the server, and optionally wherein at least one cache data source is a memory resident database that is a portion of a disk resident database and the size of the memory allocated for the memory resident database can be set by the user of the computer system, or can be automatically allocated by the system according to pre-set protocols.

16. The computer system of claim **15**, wherein the at least one cache data source is adapted to cache at least one member selected from the group consisting of data from a data source, a source structure map, an object or object definition, metadata, object schema, an object schema, and an XML file which defines or describes a member selected from the group consisting of data from a data source, a source structure map, an object or object definition, metadata, and an object schema.

* * * * *

EXHIBIT G

US007167862B2

(12) **United States Patent**
Mullins

(10) **Patent No.: US 7,167,862 B2**
(45) **Date of Patent: *Jan. 23, 2007**

(54) **SESSION BEAN IMPLEMENTATION OF A SYSTEM, METHOD AND SOFTWARE FOR CREATING OR MAINTAINING DISTRIBUTED TRANSPARENT PERSISTENCE OF COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS**

(76) Inventor: **Ward Mullins**, 2222 Leavenworth St., Apartment 304, San Francisco, CA (US) 94133

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 491 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **10/386,011**

(22) Filed: **Mar. 10, 2003**

(65) **Prior Publication Data**

US 2004/0181542 A1 Sep. 16, 2004

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.** **707/10**; 707/104.1

(58) **Field of Classification Search** ..... 707/1, 707/3, 10, 102, 103 R, 103 Y, 104.1; 705/1, 705/8; 709/219, 246; 713/200; 715/573
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,633,889 B2 * 10/2003 Dessloch et al. ..... 707/103 Y

* cited by examiner

Primary Examiner—Apu M. Mofiz

(57) **ABSTRACT**

The invention provides systems, methods and software for creating or maintaining distributed transparent persistence of complex data objects and associated data stores. In one aspect, the invention also relates to an application programming object capable of creating or maintaining distributed transparent persistence of data objects or data object graphs without the necessity of inserting any byte codes or modification of the object graph. Virtually any java object or object graph can be transparently persisted. Further, copies of a data graph or of a portion of the data graph can be automatically reconciled and changes persisted without any persistence coding in the object model.

17 Claims, 2 Drawing Sheets

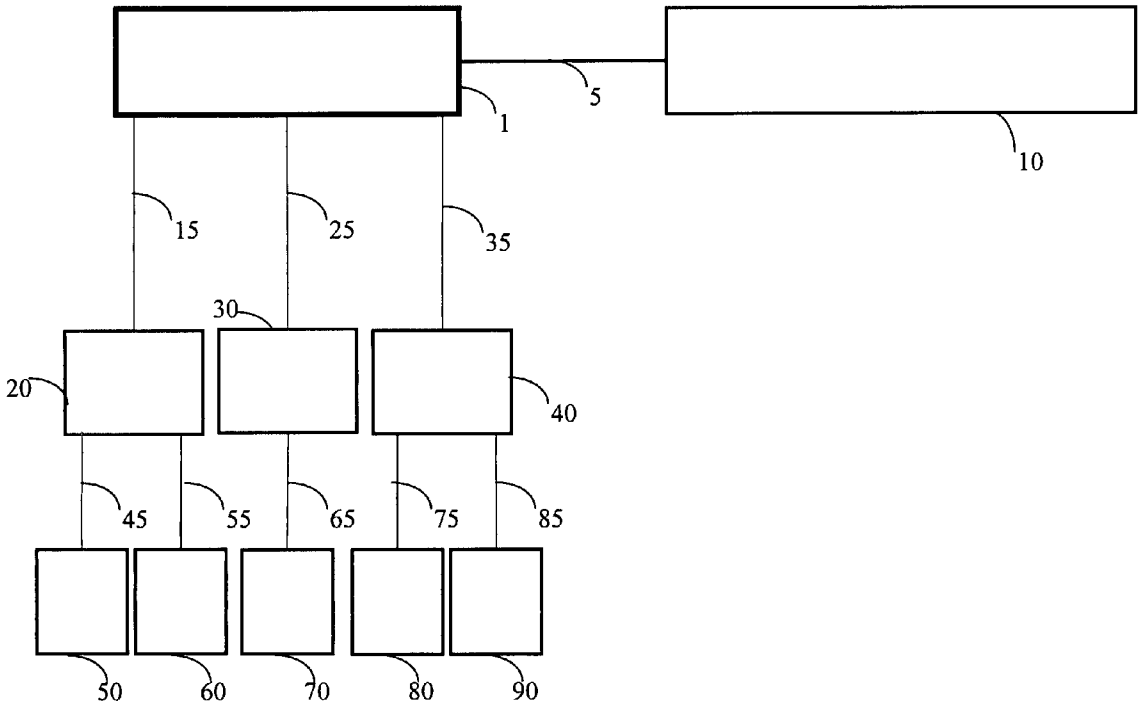


FIGURE 1

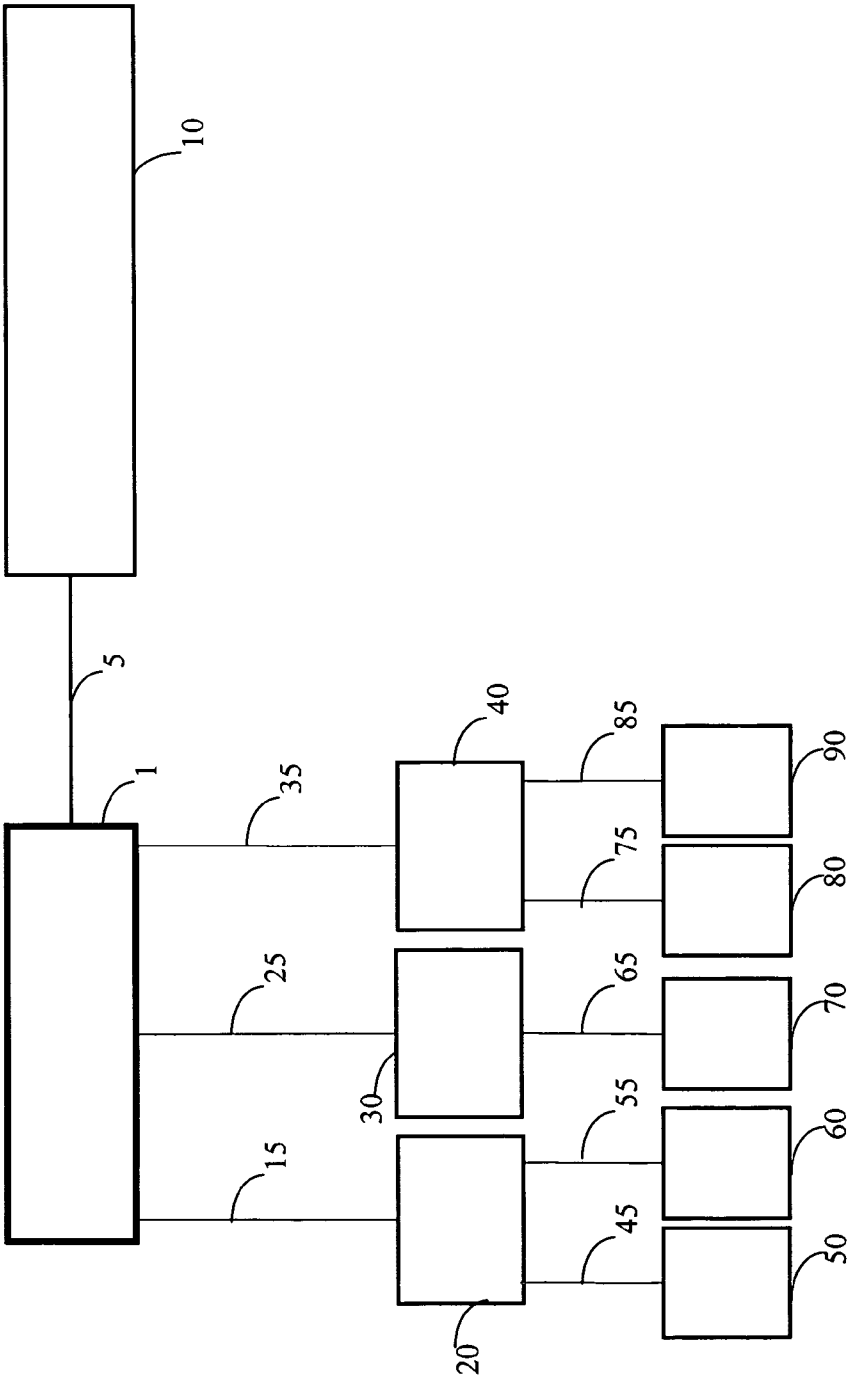
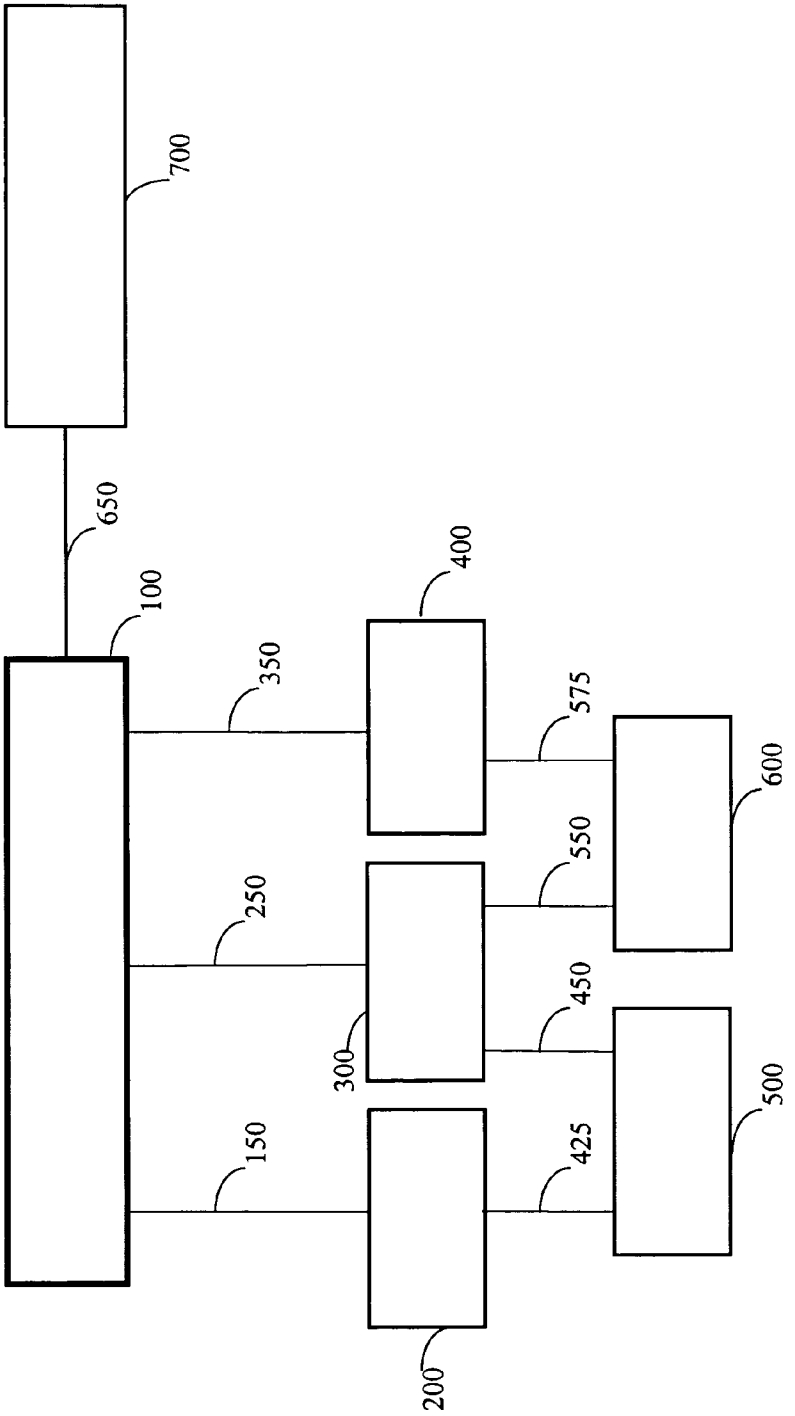


FIGURE 2



SESSION BEAN IMPLEMENTATION OF A SYSTEM, METHOD AND SOFTWARE FOR CREATING OR MAINTAINING DISTRIBUTED TRANSPARENT PERSISTENCE OF COMPLEX DATA OBJECTS AND THEIR DATA RELATIONSHIPS

FIELD OF THE INVENTION

The field of the present invention relates generally to computer systems, computer data stores and to methods and software for accessing and utilizing data stores. More particularly, the present invention relates to a system, methods and software for creating or maintaining distributed transparent persistence of complex data objects and associated data stores. In one aspect, the invention also relates to an application programming object capable of creating or maintaining distributed transparent persistence of data objects or data object graphs without the necessity of inserting any byte codes or modification of the object graph. Virtually any java object or object graph can be transparently persisted. Further, copies of a data graph or a portion of the data graph can be automatically reconciled and changes persisted without any persistence coding in the object model.

BACKGROUND OF THE INVENTION

Systems for accessing data stores from object oriented languages have been used for many years. A frequent approach to accomplish access of data stores involves writing and embedding custom access code within an object application needing the access. This approach is generally limited to having the custom code access only a single relational table within a relational database or similar construct within any other data store (hereinafter collectively "data store"). Under the circumstances where a developer has control over the design and creation of a data store from its inception, it is possible to design and store meaningful information in a single table. Such design opportunities are usually rare, however.

Generally, the methods for producing persistence for a data object, complex data object or a data store conflict with the goals of producing pure object application models where the object models do not include persistence objects or persistence byte code. Particular difficulties exist in a distributed environment since an object application model may exist in one or more of a computer's memory, an application data store and an application information storage repository that may be independent of the data store organization or object definitions. Advancements in the art have been made with respect to tools for conveniently mapping objects to systems of tables and maps in order to expedite accessing, changing and updating data stores. See, for example, U.S. Pat. No. 5,857,197 (and its associated programming interfaces ("APIs")) describes tools for translating object data to relational data, relational data to object data, and object data to object data to expedite the use of data stores. The BMP and the CMP Installer portions of CocoAdmin tool in the CocoBase™ Enterprise for O/R Binary Software (Thought, Inc. 657 Mission Street Suite 202, San Francisco, Calif. 94105 http://www.thoughtinc.com,) provide means for providing persistence in the EJB environment.

Persistence problems arise with the creation, access, changing or deleting of an object application model that utilizes such data stores. The object application model may be distributed over multiple physical computer machine

locations or even distributed over multiple Internet website locations that may be independent of the data stores. The object application model may utilize a different set of data objects or different set of definitions for relationships between data objects than that of one or more of its data sources. In most situations, the respective structures of the data sources and of the object applications model simply do not conveniently allow for mapping, accessing or changing of an overall schema of application data objects as well as any associated definitions of relationships between two or more data objects or elements within a data object.

Importantly, relationships may exist between a data object and one or more of the other data objects found in the object application model or in a data object of the data source. A relationship between one data object and another data object or with a data source may be member selected from the group of three relationship types consisting of 1 to 1(1-1), 1 to many (1-M) or many to many (M-M). Complex combinations of these relationships may exist as a data object relationships definition for a given data object. These relationships are described or illustrated in further detail later in this document.

Objects may logically span multiple relational tables or multiple object databases, and may even be distributed over a logical (or hypothetical) computer system involving multiple physically independent computer systems or even multiple website locations. Creating, accessing, maintaining or updating an object application model can require working with multiple translation modules and require tedious and repetitive updating of multiple individual computer systems or multiple data sources in order to do useful work and keep the object application model synchronized. Such approaches are both costly and unwieldy in terms of computing and development resources, particularly with respect to Internet based electronic commerce (eCommerce) object application models.

Data objects of an object application model are often a feature of eCommerce object programming applications, where information is obtained from a data source and the data is defined as a data object (e.g., as a Java class) for use with another computer application. In practice, a data object or model of data objects may exist only in the random access memory of a computer memory system, or may be saved to either a data source or to some other type of retrievable information repository. A programmer or administrator of an object data application cannot easily access or track the overall model or diagram of data objects for an object application model or some of its specific elements. Unfortunately, tools for accessing and persisting data objects and associated data object relationships of a complex data object graph model have not been well implemented in the field of object language programming.

A computer application can execute one or more of the following non-limiting actions with respect to one or more of the members selected from the group consisting of data, a data object, and a data object definition: access data, change data, create data, create a new relationship between one or more data objects by creating or changing at least one data object relationship definition, change or delete a relationship between one or more data objects by changing or deleting at least one data object relationship definition, access a data object relationship definition and use its parameters to access a data source or a data object, and access one or more data object relationship definitions or data objects to create a new data object or data object relationship. Any changes executed by a computer application with respect to one or more of the members selected

from the group consisting of data, data object or data object definition may need to be properly persisted (permanently stored) to preserve any changes to one or more of the members selected from the group consisting of data, a data object and a data object definition.

A data object and an associated data object relationship definition may be represented by a complex data object graph ("CDOG"). A CDOG, for the purposes of this document, may be thought of as a computer program data object graph that represents a data object having at least one relationship with at least one other data object or with itself via a circular link. When the data object of a CDOG is implemented in the Java computer program language, the CDOG may be further defined as being a Java Data Object Graph ("JDOG").

There are needs for software, methods and systems that can easily detect and persist any changes to at least one member selected from the group consisting of a data object, any data associated with the related object, or any associated CDOG definition (i.e., an changes to the data object, data or to a relationship of the data object with another data object). For example, there is a need to be able access a pure object model definition from a repository based O/R mapping tool or from a modeling tool repository and provide persistence for the object model without inserting any byte code or additional objects into the object model.

Accordingly, there is a strong need in the art for a computer applications programmer tool designed to assist a programmer or administrator in the actions of providing persistence for data objects or data object graphs when deleting, inactivating or updating a CDOG, wherein the computer applications programmer tool can be configured to automatically reconcile all or a portion of a CDOG and copies thereof on a distributed environment when data objects or relationships are inserted, deleted, inactivated or updated for a CDOG. A particularly strong need exists for such a tool having the further ability to be configured to persist, propagate and reflect system wide (in a local or distributed computer system) any such changes to a CDOG instance to all instances of the CDOG and to all instances of associated data, data objects and data object relationships.

Definitions

The following non-exhaustive list of definitions is used herein to define terms that may otherwise be confusing or can sometimes have multiple meanings. Each occurrence of a defined term in the above text, in the text that follows, or in the claims of this document, is to be given the meaning ascribed to it in the list of definitions below.

"Instance" as referred to in this document in the context of computer software applications is a single occurrence of a software logical element in the memory of a computer system, such as a "class", an "object", a "data object", and the like. This is analogous to the occurrence of a logical memory unit representing a row of data in common practice.

"Class" as referred to in this document in the context of computer software applications is a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). In practice, a class is a logical unit used as a logical template in an object oriented language from which to allocate new instances of objects.

"Object" as used in the context of this document is a general term referring to a logic unit in a computer application or a computer software program where the application or program is based upon an objected oriented programming language (e.g., Java). The term "object" may ordinarily be used interchangeably with the term "class" as a template or as an instance depending on the context.

"Data object" as referred to in the context of this document represents the concept of the occurrence of an object that holds data within a specific computer application domain and is likely to have its contents stored in a persistent data source of a computer system (e.g., a database server, a binary file, a text file, or even in a combination of two or more of such a persistent data sources of a computer system). A data object may exist as an independent data object without any relationship to any other data object or it may have one or more relationships with itself or with one or more other data objects.

"Complex data object" (or "CDO") as used in the context of this document refers to the occurrence of a data object that has at least one or more relationships with itself, or at least one or more relationships with one or more other data object(s). In a given instance of a CDO at least one relationship is populated as a link, as defined below. A CDO may have a multiplicity of different relationships with itself or with one or more additional CDOs.

"Relationship" or "data relationship" as used in the context of a CDO refers to the type of logical combination that occurs between a data object with itself, or refers to the type of logical combination that occurs between a data object and at least one another data object. Among other references or descriptions, such a relationship is always referred to or partially described by a "relationship type". This term is used in an object oriented language context to reference or describe any expectations, actions and limitations possible between two or more data objects.

"Relationship type" in the context of this document is a label that specifies the possible multiple combinations that can occur between a CDO and itself or with at least one other CDO. The possible relationship type labels are 1-1 (one to one), 1-M (one to many) and M-M (many to many). A given CDO may be simultaneously related to more than one other CDO through several different types of relationship. This is also sometimes referred to as the multiplicity of the relationship.

"Link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself, between a CDO and another CDO. The occurrence of at least one populated link results in an instance of the CDO. This may be considered a synonym for a "relationship" between two objects.

"Circular link" as used in this document with respect to a CDO identifies a particular occurrence of a relationship between a CDO and itself that may be direct or indirect (e.g., linked to itself through another CDO).

"Relationship definition" or "relationship description" in the context of this document and computer software applications refers to information, or an abstraction of information, regarding a "relationship", "data relationship" "relationship type" or a "link" that can be stored, accessed, transferred, communicated, displayed or edited.

"Complex data object graph" or "CDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships.

"Java data object graph" or "JDOG" is a term employed herein as an abstraction to logically represent a set of complex data objects and a set of their corresponding relationships that are part of a Java programming application.

"Application model" or simply "model" are essentially interchangeable terms employed herein as abstractions to

logically convey a collective description or other representation for a set of complex data objects and a corresponding description or other representation of their relationships. In one respect, these terms are used logically herein provide a general way of efficiently communicating when referring to set of metadata (i.e., data about data) that describes possible data entities (e.g., objects, database tables, maps, etc,) data relationship types, and data constraints involved in a computer system or application, or in a specific instance of an application. It is important to understand the context in which the terms "application model" and "model" are used in this document. Ordinarily computer engineers refer to the "model" as an abstraction rather than a specific possibility or instance of the model as applied. However, in this document for the ease of communication abstractions of the model, possible implementations of the model and instances of the model are all referred to generally as "application model" or "model". From the context of its use the term will be clear. The model is the abstract relationship between classes, wherein the CEDO is an instance (or set of instances) that express(es) the model.

"Navigation", "navigating" or "navigated" in the context of the present document refers to an action implementing at least one object to interact with a set of related objects for a certain purpose, such as creation, access, insertion, modification and deletion of an object, or of one of its relationships. It is the physical act of traversing the relationships, which might be referred to as "walking up or down the graph" in common lingo.

"Navigation model" as used herein is a special type of application model that is applied specifically to a description (or other representation) of how objects can relate to each other and what might be the expected behavior when a CDOG is navigated for a certain purpose.

"Object schema" is a term employed herein as an abstraction referring to the set of data object classes that describe the possible data objects that can be created, modified or maintained in an application, or describing an instance of a set of data object classes in an application.

"Distributed Transparent Persistence" is a term employed herein as an abstraction referring to the concept of providing persistence for a member selected from the group consisting of a data object, a data object graph, associated data and data object relationships in a distributed environment without the need for the insertion of byte code or data objects in an object model or schema.

"CocoBase Proxy Classes" is a term employed herein used in referring to wrapper classes that provide CocoBase runtime compatibility for objects that aren't inherently database aware. A computer system can persist the attributes and data for any data object that is wrapped with a CocoProxy wrapper class by simply using CocoBase facilities. For example, source code for the (attribute based) CocoProxy and (get/set method based) CocoProxyM classes are available under the thought\cocodemo3tier31\demos\pguide directory, when the CocoBase software tools suite is installed on a computer system. This optional design provides an extensible mechanism for instances of data extraction and propagation.

"CocoBase Navigation API" is a term employed herein to refer to an example of an API that provides database relationship mapping and object graph management capability for persistent objects. Database relationships are mapped to object links using CocoBase Navigator link definitions. Persistence control is provided at each class level in the object graph. Each of the Select, Insert, Update and Delete operations are individually configurable.

"CocoBase Transaction API" is a term employed herein to refer to an example of an API that provides object oriented transaction support. Transaction objects are used to persist data object attributes and maintain synchronization between database and in memory attribute values. The Transaction API has many built in optimizations, and applications utilizing CocoBase transactions generally benefit from reduced database and network overhead. This facility allows the developer to group together object changes into a single unit of work whose implementation or storage will succeed or fail all at once.

"CocoBase Factories" is a term employed herein to refer to examples of software modules and softwares libraries that are used to provide automated, custom object instantiation behavior. Factory behavior is completely customizable. For example, a factory may be used to bind newly instantiated objects to a transaction object, to load a graph of related objects using the CocoBase Navigator, or to implement polymorphism in a database result set. For example, a ProxyFactory class is part of the current CocoBase software tools suite distribution in the thought\cocodemo3tier31\demos\pguide directory, and this factory returns result set objects wrapped in a CocoProxy wrapper, when a CocoProxy wrapped key object is passed into the CocoBase runtime software module as part of a query that needs processing by the CocoBase runtime module.

"CocoBase Repository" is a term employed herein as an abstraction referring to a datasource to dataobject mapping repository and associated software modules that is installed into a datasource (or may optionally be a single stand alone file or set of files that circumscribe a set of datasource to dataobject mapping definitions and associated software modules, or may be an alternate data source). A repository can optionally be in a format such as XML, XMI and the like. See, U.S. Pat. No. 5,857,197, the CocoBaseEnterprise O/R Tools Suite, and the co-pending patent appliction entitled "Dynamic Object-Driven Database Manipulation and Mapping System" for more detailed descriptions of mapping repositories, and the like.

"CocoBase Transparent Persistence for Objects and Object Models". All models using a relational database for map storage require the CocoBase repository to be installed into the database, or in a stand-alone source accessable to CocoBase. The installation of a mapping repository can occur automatically, if required, when using CocoAdmin to log into the database. Pre-existing database tables can be used, provided that the CocoBase repository is first installed into the database, or accessible to CocoBase from an alternate location. Several example of applications that implement CocoBase transparent persistence are included in the CocoBase software tools suite distribution (see the distribution documentation for the location of folders or directories containing such examples).

SUMMARY OF THE INVENTION

An object of the present invention is to provide a system for creating or maintaining transparent persistence of a complex data object, a complex data object graph (CDOG) model, or a portion of a CDOG. In a preferred embodiment, an object of the present invention is to provide such a system that can selectively persist all or a portion of a CDOG model when the model is a member selected from the group consisting of an object model generated from a data object mapping repository and an object model generated from data

object modeling tool repository. A further object is to provide such a system is located on, or is part of, a local or distributed computer system.

An object of the present invention is to provide a method for creating, maintaining, accessing, navigating and persisting complex data objects stores in a repository. In a preferred embodiment, an object of the present invention is to provide such a method having the step utilizing the storage facilities of an enterprise EJB server to store and maintain the data object graph. In a further object, such a method involves a local or distributed computer system.

An object of the present invention is to provide a computer software component that operates in an EJB environment, or the like, wherein the component has the capacity to access an object model repository or an instance thereof in a computer memory or in another temporary computer storage store device and persist at least one action selected from the group consisting of creating, maintaining, accessing, navigating, updating or deleting complex data objects as a CDOG model. In a preferred aspect, the computer software component is an Enterprise Bean selected from the group consisting of Stateless, Stateful and Entity Beans. In a further preferred object the computer software component is an EJB Session Bean built on top of CocoBase runtime libraries having the ability to persist all or a portion of a CDOG model or instance thereof. An even more preferred object is to provide such a computer software component capable of transparently persisting all or a portion of a CDOG model or instance thereof for a local or distributed computer system and automatically reconciling and persisting any changes to an instance of the CDOG model or any changes to the repository definition for the CDOG model.

A preferred object of the present invention is to provide a software tool comprising a navigation API and software component (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

A further object of the present invention is a software tool capable of reading a source programming object logic model or a database in a format selected from the group consisting of a UML data file a XMI data file, and a XML file and converting the information into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file. In a preferred object, the software can automatically generate a transparent persistence layer that corresponds to the object model information of the source file.

A further object of the present invention is to provide a software module and source code known as a an Java entity bean (such as a generic session bean) that is capable of providing persistence of either or both of a data objects and a data model, in total or in part as determined through setting established by a user of the computer system.

BRIEF DESCRIPTION OF THE DRAWINGS

For the non-limiting purpose of illustrating some of the concepts of complex data objects CDOs, i.e., data objects and their relationships to one another, according to the invention, two CDO graph drawings FIG. **1** and FIG. **2** are provided.

FIG. **1** is a complex data object (CDO) graph drawing, which illustrates a customer object and some of its related objects (billing address, orders and items ordered), as well as relationships between the objects. Relationships of the types 1 to 1 (1-1) and 1 to many (1-M) are shown in this CDO graph. More specifically, FIG. **1** illustrates a CDO graph drawing presenting an instance of a customer object 1 having a 1 to 1 (1-1) relationship (**5**) with its customer billing address object **10**, and a 1 to many relationship (collectively **15**, **25**, and **35**) with the three outstanding order objects **20**, **30** and **40**, respectively. Order object **20** is an instance of an outstanding order object having a 1 to many relationship (collectively **45** and **55**) with the two items ordered objects **50** and **60**, respectively. Order object **30** is an instance of an outstanding order object having a relationship with a single order item, but order object **30** has a 1 to many relationship (**65**) with the item ordered object **70**, since many order items could have been associated. Order object **40** is an instance illustrates a 1 to many relationship (collectively **75** and **85**) with the two items ordered objects **80** and **90**, respectively.

FIG. **2** is a complex data object (CDO) graph drawing, which illustrates a company object and some of its related objects (corporate address object and some of its departments and employees), as well as relationships between the objects. Relationships of all three types: 1 to 1 (1-1), 1 to many (1-M) and many to many (M-M) are shown in this CDO graph. More specifically, FIG. **2** illustrates a CDO graph drawing presenting an instance of a company object **100** having a 1 to 1 relationship (**650**) with its corporate address object **700**, and a 1 to many relationship (collectively **150**, **250**, and **350**) with the three company department objects **200**, **300** and **400**, respectively. Since employees of this company may work for more than one of the company's departments, the company department objects **200**, **300** and **400** in FIG. **2** are three instances (many) of company department objects having relationships (**425**, **450**, **550** and **575**, respectively) with two (many) employee objects (respectively, **500** and **600**). The cross-assignment of employee object **500** to both company department objects **200** and **300**, and of employee object **600** to both company department objects **300** and **400**, illustrate a complex many to many (M-M) relationship of departments to employees for this company.

DESCRIPTION THE INVENTION

The present invention provides a system for creating or maintaining persistence for all or a part of a complex data object graph model when a user is creating, maintaining, accessing and navigating complex data objects as a complex data object graph model. In one embodiment, the system comprises a computer system having a user interface, a working memory storage area and at least one device for permanently storing information, and said system being loaded with at least one portion of a computer software program, wherein said software program comprises at least one user access interface and a set of programming routines designed for creating or maintaining transparent persistence when a user is creating, maintaining, accessing and navigating complex data objects as a CDOG model. In a preferred embodiment, the present invention provides such a system that can persist any part or all of a CDOG model instance, and to resolve and persist any changes to the model or to the repository for the CDOG model. Most preferred is such a system providing a point and click graphical user interface.

The present invention provides a method for creating, maintaining, accessing, navigating and persisting complex

data objects stores as a CDOG model and providing transparent persistence in a distributed environment, comprising the steps of:

a) creating or accessing a CDOG representation definition or accessing an instance of the CDOG,

b) monitoring and recording any changes to the CDOG or a portion thereof on a local or distributed computer system, and

c) persisting any changes to the CDOG on a local or distributed computer system by updating the CDOG or CDOG representation definition to reflect and changes to any copies of the CDOG or CDOG representation definition, and saving a copy of the updated CDOG representation definition to a data source or to another type of information repository.

In a preferred embodiment, the present invention provides such a method wherein the information data is a data store managed by an EJB Enterprise server. In a further preferred embodiment an EJB Enterprise Java Bean is the software component having the ability to access a CDOG repository file, an UML XMI file or an instance of the at least one portion of the CDOG in a computer's memory or on a distributed network and provide the transparent persistence for the CDOG model.

The present invention provides a computer software program having at least one user interface and having at least one feature that provides for at least one action selected from the group consisting of creating or maintaining transparent persistence when a user of a CDOG model is creating, maintaining, accessing or navigating a CDOG model. In a preferred aspect of the invention, the present invention provides a software program, or an association with a software program, having a feature for displaying, printing or displaying and printing a representation of the CDOG model as a graph, or as a set of tables representing a CDOG model. In a further preferred embodiment, such a software program has an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the complex data object CDOG model (or has both an interface for editing and an editable input or source, such as a file).

The present invention also provides an application programming interface ("API"), as a preferred embodiment, wherein the API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In one aspect the API can be accessed by an independent computer software program, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In a preferred aspect, the present invention provides such an API as part of a software tool package that provides a method for displaying or printing a representation of the CDOG model as a graph or set of tables representing the CDOG model (or providing methods for both displaying and printing a representation of the CDOG model). In another preferred aspect, such a software tool package that includes the API provides an editing interface for editing the CDOG model, or has an editable input or source, such as a file, that can be modified to implement changes to the CDOG model (or provides both an editing interface and an editable input or source, such as a file,).

A preferred embodiment of the present invention provides a software tool comprising the API according (as described above), adapted for a local network or a distributed network environment, wherein said software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGs through network APIs.

In another preferred embodiment, the CDOG API is the CocoNavigator API (or the more automated CBFacade API) which provides support for manipulating complex objects in conjunction with the THOUGHT Inc. CocoBase Enterprise Object/Relational (O/R) database (hereinafter "CocoBase") mapping tool (see U.S. Pat. No. 5,857,197 (incorporated herein in its entirety), for concepts upon which the CocoBase mapping tool is based). An object is considered to be complex when it has established or potential relationships (i.e. links) to other objects. For instance, a Company object instance may be linked to an instance of its corporate address and instances of a Department object, which in turn may be linked to a set of instances of Employee objects. An example of such an CDOG is shown in FIG. **2**, for example. The combination of these objects and their links constitute a graph of objects (CDOG model) that can be manipulated using the CocoNavigator API. Since the CocoNavigator API (or the CBFacade) works in a Java Programming language environment, this CDOG example may also be referred to as a JDOG example.

In one embodiment, the invention provides a system for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a CDOG model, comprising:

a) a set of definitions for the relationships between a data source schema and objects capable of sotring data for an object language application, wherein the set of definitions is stored in a repository;

b) a set of definitions for the relationships between objects for an object language application,

c) an object name, a list of objects, or a set of objects that are to be persisted,

d) an object or set of objects as a programming module that contains logic capable of persisting an indicated object or set of objects,

e) an input method to inform the programming module of d) with the location of information related to a), b) and c), and

f) at least one data source to which persisted data may be stored.

A preferred system as described above, is such a system that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the CDOG model. More preferred is such a system that provides persistence for a CDOG model on a distributed network environment. Even more preferred is such a system for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a CDOG model, wherein d) comprises a generic ejb stateful session bean. Even more preferred is such a system that provides persistence for a CDOG model on a distributed network environment.

In a particularly preferred embodiment, such a system as described above does not require any modifications to an object model or the inclusion of any persistence byte code in

the object model in order to provide persistence for all or a portion of the CDOG model, wherein d) comprises a generic ejb stateful session bean.

In a further embodiment, the invention provides a software module, or set of modules, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a CDOG model, comprising:

i) a software module for creating a set of definitions for the relationships between a data source schema and its stored data to objects in a object language application,

ii) a software module for creating set of definitions for the relationships between objects in an object language application,

iii) a software module providing an input means for an object name, a list of objects, or a set of objects that are to be persisted,

iv) a software module containing the programming logic to persist an object or set of objects based upon access to (a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, (b) a set of definitions for the relationships between objects for an object language application, and (c) an object name, a list of objects, or a set of objects that are to be persisted, and

v) a software module having an input method to inform iv) of the location of the information related to iv)(a), iv(b) and iv(c).

A preferred embodiment of the above invention comprises a software module, or set of modules, as described above, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a CDOG model, comprising software code for generating, accessing, or maintaining a generic ejb stateful session bean object as part of the software module of iv). Preferred is such a software module, or set of modules that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the CDOG model, comprising software code for generating, accessing, or maintaining a generic ejb stateful session bean object as iv). More preferred is such a software module, or set of modules that provides persistence for a CDOG model on a distributed network environment.

In another preferred embodiment, the invention provides a software tool comprising the API according to the invention or interfacing therewith, wherein the software is adapted for performing the following steps:

a) reading a source programming object logic model or a database definition in a format selected from the group consisting of a Unified Modeling Language ("UML") data file, an XML Metadata Interchange ("XMI") data file, and an Extensible Markup Language ("XMI") file; and

b) converting the information of (a) into a target member selected from the group consisting of a database definition XML file, a database mapping definition file, and a CDOG definition file.

In a further preferred embodiment, the present invention provides a software tool as described above that is adapted for performing at least one of the following additional steps:

a) displaying a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

b) storing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

c) printing a representation of a source programming object logic model or a database definition file in a format selected from the group consisting of a UML data file, a XMI data file, and a XML file;

d) displaying the target information of (b);

e) storing the target information of (b); and

f) printing the target information of (b).

In a still further preferred embodiment the software tool described above provides target information that is at least one member selected from the group consisting a CocoBase database definition repository file, a CocoBase map, a CocoNavigate (or CBFacade) CDOG definition, a CocoNavigate (or CBFacade) object link descriptor, and a CDOG object graph definition data file. Further preferred is such software, wherein the source is a UML/XMI document containing sufficient DTD information to exchange modeling information with a UML modeling tool.

In one embodiment of the software tool according to the invention, the software tool can be set to automatically generate a persistence layer that corresponds to the source UML class diagrams provided by an exported source file from a case tool capable of exporting UML/XMI, such as Rational Rose, Together and ArgoUML.

An easy way to understand the job performed by a CDOG Navigator API, such as the CocoNavigator API, is to imagine it as a monitor that can provide at least one object to intercept any accesses to the data objects and their relationships that constitute the CDOG, as well as any changes to the data source upon which the CDOG depends, in order to provide a means to persist any changes to the CDOG (or CDOG model) or related data in the data source. Any changes to the CDOG (or CDOG model) can then be propagated by the CDOG Navigator API to a persistent data source through a CocoBase Enterprise O/R connection. Similarly, persistent data that updates a data source can be utilized to create a CDOG model or to update a CDOG model.

Each CDOG (or CDOG model) managed by the CDOG Navigator API (or CBFacade) can be associated by the CDOG Navigator API (or CBFacade) with a CDOG descriptor (such as a file) that may be utilized to represent all or part of a "navigation model". In this respect, a navigation model may be abstractly described as essentially a namespace in which a set of data objects and a set of link descriptions (i.e., relationship types between data objects) are each defined. In a preferred embodiment, a data source, data field, data field size and data field type are listed for each data object in the set of data objects. In another preferred embodiment, at least one link description for two data objects, or for a single data object having a circular link (where the link description is utilized in conjunction with the CocoNavigator API and CocoBase mapping tool) contains one or more of the following types of information (however other related information may be utilized):

at least one link type (e.g., 1:1, 1:M, M:1 and M:M) between data objects

at least one link qualifier (currently supported qualifiers are STD and BEAN)

at least one associative CocoBase map (optional, for M:M links only)

the names of the CocoBase maps related to each of the data objects (usually two maps that are not necessarily distinct), and

information that may be obtained for each of the CocoBase maps related to the relationship between two data objects, including:

the name of a relationship link between the two data objects

the names of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects

the name of a data source for each of the fields (i.e., keys) in a CocoBase map that are used to establish a relationship link between two data objects

the names of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)

the names of a data source for each of the fields (i.e., keys) in the associative CocoBase map (optional, for M:M links only)

the name(s) of any java classes corresponding to fields that would impact upon the relationship between two data objects

a setting parameter indicating whether the a loading of a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)

a setting parameter indicating whether a deletion of information from a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s)

a setting parameter indicating whether an insertion of information into a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s), and

a setting parameter indicating whether an update of information in a particular CocoBase map that will impact upon a relationship between two data object should be cascaded to other related map(s).

Some preferred features provided by the CocoNavigator API and its associated software, when utilized with the CocoBase mapping tool, or with a similar mapping tool, are as follows:

(a) provides multiple navigation model for the same CDOG model

A preferred embodiment of the CocoNavigator API or an associated program module is configured to allow many navigation models to be used with the same set of java classes and CocoBase maps that are associated with a CDOG model. This preferred embodiment API, or an associated computer program module, may also be configured to permit a user to switch dynamically from one such navigation model to another. Thus, while a given application is being executed, it is possible to completely change the relationships between data objects and how CDOGs of a CDOG model should be managed.

(b) circular link definitions

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage circular data object links. In a navigation model, a circular link may be defined as occurring when a data object is directly or indirectly linked to itself as part of a CDOG. Such relationships can be created, accessed, supported and correctly managed via the CocoNavigator API.

(c) bi-directional and oriented links

A preferred embodiment of the CocoNavigator API, or an associated computer program module, is configured to permit a user to create, access, support and correctly manage links between data objects as either a bi-directional link or as an oriented link. In this embodiment links between data objects are usually bi-directional links and can be navigated back and forth as an un-oriented navigational model of the CDOG model. With such a bi-directional link feature activated, any object in the navigation model in a given CDOG model can be used as a navigation entry point. In an oriented link navigation model, the link may be set as an oriented link accessible from a single direction. Thus, changes must be made from top down or from bottom up on a relationship tree representation of the navigation model, and some actions may need to originate from a pre-set node of the relationship tree. In this embodiment, such types of links can be created, accessed, supported and correctly managed via the CocoNavigator API.

(d) dynamic link proxy properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, is configured to permit a user to populate any object property (e.g., public field or a getter/setter pair) having its type declared as an object class with special dynamic proxies that can monitor and update the state of a relationship link. Examples of such types of object classes are thought.Cocobase.navapi.LinkObjProxy of CocoBase and the Sun Microsystems java class, java.util.Vector. Bi-directional or single directional object references can be consistently maintained in this manner, or by other similar logical mechanisms.

(e) automatic synchronization of object properties

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to automatically merge a data object and its properties into a CDOG navigation model representation when the class of object being navigated (being created, accessed, supported or maintained) has a property (e.g., a field or a getter/setter pair) with a name matching the name of a corresponding link as defined in the navigation model.

(f) pure object models

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to include a data object as a relationship link to another data object in a navigation model without the need to declare fields and properties for the data object. Such links should be limited however to cases where no computer code dependency will exist between the java classes that will be associated with such linked data objects of the navigation model. Thus, such a navigation model can be a pure abstraction, and more reusable than just a populated version or single hypothetical instance of the abstraction.

(g) customized link visitors

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can provide an interface which can be configured to permit a visitor object to use this interface to visit (e.g., access or change) a data object that is part of a CDOG navigation model. In conjunction with Coco-

Base, an example of CocoNavigator API implementing this concept might using a LinkVisitor object (class). Other similar classes may be defined for this purpose. The visitor interface provides a way for a user to customize and adapt the software, and thereby allows a user to extend the functionality of the CocoNavigator API, or the functionality of an associated computer program module, to provide a desired customizable behavior in their CDOG navigation model.

(h) distributed environments

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to operate as a tool to create, access, support and correctly manage a CDOG navigation model in a server environment (e.g., in an EJB container) and to persist any changes to the CDOG navigation model when a navigation model is distributed across a local network or when the navigation model involves a distributed network (e.g., a navigation model distributed across internet connections). In one aspect, on the server side of the network, a CDO or a CDOG model of any complexity sent by clients (e.g., serialized copies) across a local network, or across a distributed network, can be correctly merged into a CDOG model by the CocoNavigator API, or by an associated computer program module. Additionally, the CocoNavigator API, or an associated computer program module, can be configured to send a CDO or CDOG model to a client along with link proxies serialized with parts of the CDO or CDOG model that are being monitored by such link proxies. By sending the CDO or CDOG model copy along with such link proxies and associated parts of the CDOG or CDOG model to a client, dynamic link proxies (described above in (d)) can be used by the client side. Accordingly, a CDOG navigation model can be created, accessed, supported, managed and persisted over a distributed network.

(i) group loading or lazy loading of links

In a preferred embodiment, the CocoNavigator API, or an associated computer program module, can be configured to monitor some or all of the data objects (and associated relationship definitions) participating in links of a CDOG navigation model. In a more preferred embodiment, the data objects (and associated relationship definitions) participating in links of a CDOG navigation model can be loaded as they are needed (lazy loading) from a persistent data source or from another type of information repository. This lazy loading feature can permit a very large CDOG navigation model to be loaded in a per-link basis as needed. In a more preferred embodiment, the lazy loading feature can be configured to prevent a link from being loaded twice and can be configured to detect cycles of related loads that have been previously loaded. Such configuration adaptability can be exploited to provide the more efficient use of the resources for a computer system or network or for a distributed computer network.

Some examples of preferred features that can be provided by the CocoNavigator API and its associated software, which are particularly enhanced when utilized with the currently existing CocoBase mapping tool, are as follows:

(aa) CocoBase transactions

A preferred embodiment of the CocoNavigator API or an associated program module is configured to bind a data object or data object relationship of a CDOG navigation model with a CocoBase transaction. Such configurations may be utilized to optimize access to the database server and to enhance performance of the computer system, or systems, involved with the CDOG navigation model.

(bb) non-CBProp objects

The CocoNavigator API or an associated program module can be configured to automatically detect non-CBProp objects (objects with classes that do not implement the CocoBase interface known as the CBProp interface), and automatically create proxies in order to persist such objects as part of a CDOG navigation model.

(cc) virtual foreign key fields

The CocoNavigator API or an associated computer program module can be configured to use proxy classes of CocoBase, such as the CocoProxyM classes, when appropriate. Implementing proxy classes such as the CocoProxyM classes can provide a system for creating, accessing, supporting, properly maintaining and persisting virtual foreign key fields (i.e., foreign key fields do not need to be defined in the object class for Java programming implementations) by the CocoNavigator API, or by an associated computer program module.

(dd) transparent persistence

The CocoNavigator API or an associated computer program module can be configured to use a configurable network component capable of acquiring and persisting CDOGs through network APIs and thereby provide persistence transparently with respect to the applications that are using the CDOGs. In a preferred implementation, there is provided a software tool comprising the CocoNavigator API that is adapted for a local network or for a distributed network environment, wherein the software tool provides persistence in an object oriented language environment transparently by implementing a configurable network component capable of acquiring and persisting CDOGS. Such a concept may be referred to as transparent persistence in the context of this API, because persistence can be obtained without requiring reconfiguring of an existing CDOG model or software application in order to provide such persistence to a computer software application.

(ee) transparent persistence with automated APIs that may be further optionally utilized with a local or distributed implementations such as a session bean implementation

The CBFacade or other modules containing high level API's can be utilized that can function in place of the CocoNavigator API (or coordinate with it) to automate many of the transparent persistence functions. Such a software module or modules can be utilized in concert with a local or distributed implementation such as a session bean to provide an easier and quicker establishment of transparent persistence. The CBFacade can provide the functions as defined in (dd) and the session bean can provide a communication link between modules such as a transaction API and the navigation API (and/or with the CBFacade) to assure that transparent persistence occurrs as

required. Such transparent persistence in the context of this API can be obtained without requiring reconfiguring of an existing CDOG model or software application in order to provide such persistence to a computer software application.

From the above description of features of the API, and the features of the CocoNavigator API programming routines described below, an object computer language programmer can produce an API having the functionality of the CocoNavigator API described herein. Essentially the CocoNavigator API has an at least one user access interface, at least one data source access interface and at least three main programming modules consisting of the following programming modules or routines:

I. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:

a) a computer programming sub-routine or sub-module for obtaining from a member selected from the group consisting of a data source, an information repository, and an input device, sufficient information to construct a CDOG model containing one or more CDOGs,

b) a computer programming sub-routine or sub-module for constructing and loading into the memory registers of the computer system a CDOG or a CDOG model representation definition from a data source or other repository, and

c) a computer programming sub-routine or sub-module for sending a copy of some portion or all of the CDOG representation definition to a user or to a software program module on a local computer system or on a distributed network;

II. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or to a software program module of the computer system:

a) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system that has accessed or changed a portion of a CDOG or CDOG model, which is included in the CDOG, or CDOG model, representation definition of (I), above, and obtaining any changes to the CDOG or CDOG model,

b) a computer programming sub-routine or sub-module for monitoring a user or a software program module on a computer system who has obtained a copy of any portion of the CDOG, or CDOG model, representation definition, and for obtaining a copy of any changes that the user might have made to any portion of the CDOG, or CDOG model, representation definition, and

c) a computer programming sub-routine or sub-module for comparing a copy of a CDOG, CDOG model, or a representation definition of either the CDOG or CDOG model, to an original stored version of the CDOG, CDOG model, or an original stored representation definition for the CDOG or CDOG model, and for updating the original to incorporate any changes to a CDOG or a representation definition that are made by the user or by a software program module; and

III. A programming module or routine constructed to operate on a computer system and to provide the following features to a user or software program module of the computer system:

a) a computer programming sub-routine or sub-module for storing a new CDOG or CDOG model, or storing a definition of either the CDOG or CDOG model, in a data source or in another information repository, and

b) a computer programming sub-routine or sub-module for persisting (saving to permanent storage) either a changed portion of an updated CDOG, an updated CDOG model, or an updated definition representation for either a CDOG or a CDOG model, to a data source or to another type of information repository.

In a more preferred embodiment, the CDOG API according to the invention, (a preferred embodiment is the CocoNavigator API) can be written with an interface that accesses and uses functionalities of the CocoBase mapping tool, associated programming routines, or associated class libraries. Both the object code and the source code of the CDOG API are contemplated as part of this invention.

In another preferred embodiment, the above described CDOG API can be accessed to create, maintain, access, navigate and persisting complex data objects as a CDOG model. In a particularly preferred aspect, the API can be accessed by a computer software program such as the CocoBase mapping tool, or by a computer software program module that is part of a software package including the API, to provide at least one action selected from the group consisting of creating, maintaining, accessing, navigating and persisting complex data objects as a CDOG model. In an even more preferred aspect, the present invention provides such an API as part of a software tool package having a feature or method for displaying or printing a graph or set of tables representing a CDOG or a CDOG model. In yet another preferred aspect, such a software tool package including the API provides an interface for editing a CDOG, a CDOG model, or for editing a representation definition for a CDOG or a CDOG model to provide a modification that implements a change to a CDOG, or to its definition.

As described above, a software component according to the invention that is capable of persisting all or a portion of a CDOG may be an Enterprise Java Bean selected from the group consisting of Stateless, Stateful or even Entity Beans. CocoBase Enterprise Beans (CBEBs) are a preferred embodiment of the invention and constitute a set of Enterprise Java Beans build upon or accessing CocoBase Runtime and EJB APIs to provide persistence in a generic, efficient and transparent way. Distributed transparent persistence can be provided without the need for any object model changes or byte code modifications. CBEBs can be directly deployed on an EJB server with or without customized settings and are capable of persisting virtually any Java Object, any Java Object graph or any portion of the Object graph. The Java Objects of any complexity that are arbitrarily created by client applications can be persisted, and the corresponding object classes can be properly mapped to a CocoBase repository or other object repository such as a UML XMI repository. Copies of the Java Object, any portion of an Object Graph, or all of an Object Graph can be automatically reconciled and the changes can be persisted to a storage repository.

The CocoBase Navigation API maps database foreign key—primary key relationships to object references. The relationship maps (sometimes referred to herein as navigation models), like the database maps, are stored in a special CocoBase repository, or may be converted into an XML file (or files) and stored in a stand-alone file or directory. The Navigation API supports 1 to 1, 1 to M, and M to M links in both the object and relational spaces. For M to M

relationships, an associative table and a corresponding associative map are required, since relational databases do not directly support M to M relationships.

A navigation model must be created, or properly reference, before CocoBase links can be defined. A navigation model is a namespace used to categorize and store a set of link definitions. Generally, a single navigation model is sufficient for simple applications. Multiple navigation models are useful if an application switches between several relationship configurations. This allows different views of complex object data for the same set of underlying database tables. Having relationship mapping separated from table mapping is unique to CocoBase. Since maps can be used with multiple link models, it provides reusability at the mapping level.

Below is an example of how a client application on an EJB Server could connect to and use at least on CBEB to provide transparent persistence, in a local non-distributed environment.

EXAMPLE 1

```
//Set up the connection class
CBFacade cbf=CBFacade.create("thought.CocoBase.CBFacadeLocal");
Properties props=new Properties( );
props.put("cocosource.name",thought.CocoBase.CocoPowderPlugin20);
props.put("cocosource.jdbcdriver","oracle.jdbc.driver.OracleDriver");
props.put("cocosource.url","jdbc:cocoprop=jndiurl=java:comp/env/jdbc/sfcmbhome");
props.put("cocosource.user","SCOTT");
props.put("cocosource.password","TIGER");
props.put("cocosource.navmodel","CompanyExample");
cbf.connect(props);// create a complex object having myObj as the root of the tree
//create an example object
MyObject myObj=new MyObject( );
myObj.setSomeField(someValue);
//now ask the Facade to load the object that matches the set fields (QBE)
myObj=cbf.load(myObj);
//create an example object
myObj.setSomeOtherField(someOtherValue);
//now ask the CBFacade to save the changed values
cbf.save(myObj);
//do some more changes to myComplexObject and save it again
cbf.save(myObj);
```

Implemention of Examples 2–4, Below, Collectively Provide Persistence Via a Session Bean as Described Generally by the Following Text.

The CBSession EJB

The CBSession EJB is a Stateful Session Bean that works as an abstraction layer for CocoBase runtime classes. It provides generic transparent persistence to Java objects of any complexity in a simple yet elegant way. It is designed to run in any Java Application Server, thus combining CocoBase transparent object persistence capabilities with most features provided by such servers, such as transaction management, security, resource pooling and so forth.

The CBSession bean is very simple and easy to use. Here is an illustration of how the session bean of Examples 2–4, below might be implemented and used:

```
. . .
Context ic=new InitialContext( . . . );
Object ref=ic.lookup("CBSession");
CBSessionHome home=(CBSessionHome)PortableRemoteObject.narrow (ref,CBSessionHome.class);
String cocourl="CBSession;cocorep=configresource:/resources/MyConfigFile.properties:
cocoprop=primarykey.lookup=false,
dynamic.querying=true,jndiurl=java:comp/env/jdbc/demoPool"
//connects to a CBSession
CBSession cbSession=null;
try {
cbSession=home.create(cocourl,"MyNavModel",null);
cbSession.setDebugLevel(1);
}catch (Exception e) {
System.out.println("Could not open CBSession due to "+e);
return;
}
MyComplexObject obj=new MyComplexObject( );
. . .
//populate values for querying
obj.setValues(. . .);
. . .
//load object graph
obj=(MyComplexObject) cbSession.load(obj,"MyMap");
. . .
//do some changes to obj graph
. . .
//save changes
cbSession.save(obj,"MyMap");
```

See CBSession interface for more details.

Deploying the CBSession bean

Requirements:

coco_home/classes and coco_home/demos must be in your App Server classpath.

Ant 1.2 or later must be installed in order to compile/regenerate deployable jar files.

The Ant script build.xml will generate a deployable jar file with the CBSession bean for a particular server. Make sure the entry in the beginning of the build.xml file has the entry

<property name="cocohome"**0** value="c:/thought/cocodemo3tier31"/>correctly specified.

Targets currently defined include WebLogic 6.1, JBoss 2.4 and a generic jar file. Each subdirectory corresponds to a server, containing the standard descriptor and proprietary descriptors for that server. These descriptors can be modified in case it is required to do so.

For example, the jar file for JBoss can be generated as follows:

os-prompt>ant jboss

See build.xml for available targets.

IMPORTANT: In order to have CBSession working for a particular application model, all the required java classes and resources (i.e. the map pre-load configuration properties file, the navigation model properties file) must be in the classpath (for example, these classes and resources could be copied to the coco_home/demos/resources directory, provided this directory is included in the app server classpath). Another alternative would be to add all needed classes and resources to the CBSession bean jar file. This can be done simply by copying these classes and resources to the directory corresponding to the target server and running the ant script again. For example, for JBoss, the following directory structure would generate a jar file with all classes and resources:

```
cbsession
  . . .
  jboss
    ejb-jar.xml
    jboss.xml
    resources
      MyConfigFile.properties
      MyNavModel.properties
    my
      app
        package
          AppClass1.class
          AppClass2.class
          AppClass3.class
          . . .
```

Examples 2–4, source code for CBSession Bean and its use are as follows:

EXAMPLE 2

```
Source Code File Name=CBSession.java
package thought.CocoBase.ejbs.cbsession;
import java.rmi.RenioteException;
import javax.ejb.EJBObject;
/**
* CBSession remote interface
* <p>
* Copyright (c) 2001 THOUGHT Inc. All Rights Reserved.
*
* @author Alexandre Jonatan Martins
* @version 0.9, August 2001
*/
public interface CBSession extends EJBObject {
  /**
  * Invokes <code>load(Object,String)</code>sending
  *  <code>object.getClass().getName()</code>as the <code>mapName</code>
  * @param obj the example object
  * @return the loaded object
  */
  public Object load(Object obj) throws RemoteException;
  /**
  * Loads an object from the database
  * @param obj the <i>example<i>object
  * @param mapName the map name corresponding to the class of obj
  * @return the loaded object
  */
  public Object load(Object obj, String mapName) throws RemoteException;
  /**
  * Invokes <code>save(Object,String)</code>sending
  *  <code>object.getClass().getName()</code>as the <code>mapName</code>.
  * @Param obj the example object
  */
  public void save(Object obj) throws RemoteException;
  /**
  * Saves an object to the database
  * @param obj the <i>example<i>object
  * @param mapName the map name corresponding to the class of obj
  * @return the loaded object
  */
  public void save(Object obj, String mapName) throws RemoteException;
  /**
  * Invokes <code>delete(Object,String)</code>sending
  *  <code>object.getClass().getName()</code>as the <code>mapName<code>.
  * @Param obj the example object
  */
  public void delete(Object obj) throws RemoteException;
  /**
  * Deletes an object from the database
  * @param obj the <i>example</i>object
  * @param mapName the map name corresponding to the class of obj
  */
  public void delete(Object obj, String mapName) throws RemoteException;
  /**
  * Returns app server session debug level
  *  @return <code>true</code>if debug is on; <code>false<code>otherwise
  */
  public int getDebugLevel() throws RemoteException;
  /**
  * Sets app server session debug level as follows:
  * <p><code><pre>
  *  1 - prints basic session response
  *  2 - prints navigation info + level 1 info
  *  3 - prints detailed navigation info + level 2 info
  *  4 - prints cocobase transaction info + level 3 info
  *  5 - prints cocobase driver and jdbc info + level 4 info
  * </pre></code></p>
  * @param debug the debug level
  * @param mapName the map name corresponding to the class of obj
  */
  public void setDebugLevel(int level) throws RemoteException;
}
```

EXAMPLE 3

```
Source Code File Name=CBSessionHome.java
package thought.CocoBase.ejbs.cbsession;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.util.Properties;
/**
* CBSession home interface
*<P>
* Copyright (c) 2001 THOUGHT Inc. All Rights Reserved.
*
* @author Alexandre Jonatan Martins
* @version 0.9, August 2001
*/
public interface CBSessionHome extends EJBHome{
/**
* Creates a CBSession. Properties will be read from the evironment entries in
* the ejb descriptor. The following properties are recognized:
```

```
*<p><code><pre>
* cocosource.url=cocobase_connection_url_string
* cocosource.name=cocobase_driver_name_string
* cocosource.user=user_name_string
* cocosource.password=user_password_string
* cocosource.autoclose="true" | "false"
* cocosource.autotrans="true" | "false"
* cocosource.navmodel=navigation_model_string
*</pre></code></p>
*/
public CBSession create( ) throws RemoteException, CreateException;
/**
* Creates a CBSession. The behavior of the session data source can be customized by
* a properties object containing specific configuration details. The following
* properties are recognized:
*<p><code><pre>
* cocosource.name=cocobase_driver_name_string
* cocosource.user=user_name_string
* cocosource.password=user_password_string
* cocosource.autoclose="true" | "false"
* cocosource.autotrans="true" | "false"
*</pre></code></p>
*@param cbUrl the CocoBase connection url
*@param navModel the name of the navigation model (if not provided, uses default navigation model)
*@param props configuration properties
*/
public CBSession create(String cbUrl, String navModel, Properties props) throws RemoteException, CreateException;
```

EXAMPLE 4

In "Example 4", the source code file name is "CBSessionBean.java". This Example is contained on an attached CD-ROM under the file name "Example 4", and is incorporated herein by reference.

The CBFacadeRemote is a Wrapper API that works with the CBSession EJB (described in detail previously) It provides a high level API that simplifies the task of remote programming and distributed usage in a very elegant API. It is compatible with an interface which makes it possible to swap out Local and Distributed implementations at will, and at runtime. It is designed to run in any Java Application Server, thus combining CocoBase transparent object persistence capabilities with most features provided by such servers, such as transaction management, security, resource pooling and so forth.

EXAMPLE 5

In "Example 5"the source code file name is "CBFacadeRemote.java" and is contained on an attached CD-ROM, under the file name"Example 5", and is incorporated herein by reference.

The present may be embodied in specific forms other than those particularly described above or illustrated by the appended drawings. Upon viewing the present application preferred embodiments and other descriptions herein of the present invention, variations and other implementations that do not depart from the spirit and scope of the present invention will be apparent to one of routine skill in this field. Such variations and other implementations are considered part of the present invention and within the scope of the appended claims. Accordingly, reference should be made to the appended claims, rather than to the forgoing specification and drawings, as indicating the scope of the present invention.

From SQL to Transparent Persistence using CocoBase Enterprise OR

This text is a brief overview of how to create Java applications that use CocoBase Transparent Persistence features starting from SQL database tables. Such Transparent Persistence can be run locally on a system instead of in a distributed environment. A local application server or a JSP server such as TomCat can be utilized to connect a JSP to a database and provide persistence of objects in the model or database.

Table Mapping

Suppose a set of database tables was defined as follows:

```
CREATE TABLE COMPANY(
       ID NUMERIC,
       NAME VARCHAR(50),
       PRIMARY KEY(ID))

CREATE TABLE DEPARTMENT(
       ID NUMERIC,
       NAME VARCHAR(50),
       COMPANY NUMERIC,
       PRIMARY KEY(ID),
       FOREIGN KEY(COMPANY)REFERENCES COMPANY(ID))

CREATE TABLE EMPLOYEE(
       ID NUMERIC,
       NAME VARCHAR(50),
       SALARY REAL,
       DEPARTMENT NUMERIC,
       MANAGER NUMERIC,
       PRIMARY KEY(ID),
       FOREIGN KEY(DEPARTMENT)REFERENCES DEPARTMENT(ID))
```

The steps to create a CocoBase Map from a given table are detailed below.

1) Connect to the CocoBase Repository

Launch CocoBase and go to **File->Open New DB Connection** to open the login dialog.

In the dialog, select the JDBC driver/user/password to be used to connect to the CocoBase repository and click on the **Login >** button presented in the dialog box or window.

2) Create Map from Table

If login was successful, a dialog box will appear indicating a successful login and providing a number of task options and/or a tool bar with pull down windows.

Check the box **Generate New Map From Table** in the dialog box or window and click on the button **Launch**.

A window will appear with a tree indicating connections and tables that are available.

Expand the branches on the tree in the dialog to see the tables available in the database and then select one of the tables. Click on the button **Next >** to go to the window with the Map field details, verify if the information is correctly matching the table definition and that the Map name is appropriate. The Map can be modified at this point by changing the names or types of the fields and inserting or deleting fields.

Click on the button **Finish** to create the new Map from the selected table. A dialog will be prompted asking to confirm the creation of the Map. Click on the button **Continue Map Create** to confirm the creation of the new Map. A message will be prompted confirming that the Map was successfully generated and is already in the CocoBase repository.

Click on the button Ok button of the pop-up dialog to continue. This procedure must be repeated for all the tables containing information relevant to your application. To start again in the Map creation Window of the earlier present dialog (or at the top of the tool bar on the interface), select the menu item

File -> New Map from Table.

3) Verify the results

Go to **File->Edit Map** in the dialog to open the Map editor and verify if the Maps were created accordingly.

Relationship Mapping

Once all necessary database tables are mapped, it is possible to create Link definitions between the Maps so that 1-1, 1-M and M-M relationships between objects can be transparently managed by CocoBase. The steps to create a Link definition are detailed below.

1) Choose Link Model

Before creating a Link, it is necessary to have a Link model created. A Link model is a set of Link definitions. Multiple Link models are useful when an application needs to switch between several relationship configurations, since it allows different views of complex object data over the same set of underlying tables. The idea of having relationship mapping separated from table mapping is unique of CocoBase. It provides reusability at the mapping level, since a given Map can be reused across multiple Link models.

Usually, one single Link model is sufficient for a simple application. To create a link model, right click on the **Links** branch in the Map editor tree. A pop-up menu like the following will appear giving options such as **New Model**, **Import Model**, **Save**, **Save to location**, **Reload**, and the like.

Select the option **New Model...** and enter the name of the new Model in the input dialog that will appear next.

The new Link model will be inserted as a sub-node in the editor tree.

2) Create Link Definition

In order to create a new Link for that model, right click on the model to activate the model pop-up menu and select the menu item **New Link** in the dialog.

After clicking on the menu, a new empty Link definition will be inserted in the selected model. Expand the model branch and click on the new Link definition to see its details in a detail dialog.

Although totally unconstrained, a Link definition would normally be created for each foreign-primary key relationship in the database tables or, in the case of many-to-many relationships, for each associative table.

A Link definition is basically a combination of two roles, each having the following information:

Map Name – the name of the Map participating in the relationship
Link Name – the name of the link that connects to the Map in the opposite role. It normally matches the name of the object property (field or get/set method pair) that holds instances of the related objects.
Key Fields – the comma-separated list of fields defined according to the following guidelines:

- If this role is on the "one" side of a one-to-one or one-to-many link, this is the list of the fields in the map that correspond to the primary key defined in the underlying table
- If this role is on the "many" side of a one-to-many link, this is the list of the fields in the map that correspond to the foreign key defined in the underlying table
- If this is a role of a many-to-many links, this is the list of the fields in the map that correspond to the primary key defined in the underlying table

Java Class – the full qualified name of the Java class that will be instantiated when the relationship is loaded from the database. This is normally a class corresponding to the Map specified in the opposite role.
Cascade Load – indicated if this role is going to load related object instances.
Cascade Insert – indicated if this role is going to insert related object instances.
Cascade Delete – indicated if this role is going to delete related object instances.
Cascade Update – indicated if this role is going to update related object instances.
M:M Key Fields – This is relevant only for many-to-many Link definitions and is the comma-separated list of the fields in the associative map that refer to the Key Fields of this role. In this case, the name of associative map must be specified in the box **M:M Map Name**.

Examples of Link definitions are given in published CocoBase literature. Click on the appropriate boxes to edit the Link definition for the model being defined. Repeat this procedure for each table relationship that is relevant to the application being developed.

3) Save Link Definitions

Once Link definitions are complete, select the branch **Links** in the editor tree right and click on it to access the pop-up menu. Click on the option **Save** to save your work.

Note that unlike CocoBase Maps, Link models are not kept in the CocoBase repository in the current release. Instead, for each Link model, CocoBase will create a property file called *modelname.properties* containing all the Link definitions for that model. This property file is stored under the demos/resources directory (by default - although this can be overridden). As long as the property file describing the Link model is in the classpath (either directly or in a subdirectory called *resources*), CocoBase runtime classes will load the model and register the navigation information.

CocoBase Transparent Persistence

In the previous sections show above in this appendix, it was demonstrated how to map database tables and relationships using CocoBase. In this section, it is shown how CocoBase runtime classes can be used to provide transparent persistence to a Java Object Model which corresponds to the created Maps and Links.

Transparent Persistence with CocoBase Maps and Models

CocoBase accomplishes transparent persistence with java object models without using bytecode manipulation, proprietary interfaces or class hierarchy intrusion. This means that no special classes or interfaces are needed in the Object Model in order to do persistence with CocoBase. The only requirement is that they must have a default constructor with no arguments.

There are 3 basic Runtime components that are involved in the transparent persistence of objects:

- The CocoBase Runtime O/R mapping class that wrappers the JDBC driver and issues queries and does the actual persistence calls. This is a class such as `thought.CocoBase.CocoPowder` or `thought.CocoBase.CocoPowderPlugin20` (for jdbc 2.0 connections).
- The `thought.CocoBase.Transaction` object that can track changes of instances, and acts as a change 'buffer'. If a Transaction object is used, then it only calls a CocoBase Runtime driver - O/R mapping runtime class - when the txn.commit() is called.
- The `thought.CocoBase.navapi.Navigator` object that can track and detect changes in relationships based on Link definitions.

The `Navigator` class can function in conjunction with a `Transaction` object or it can function standalone. While complex object graphs can be transparently managed directly by the Navigator and without the `Transaction` object, the use of the `Transaction` object is generally preferred because of its buffering and update optimizations which only persist those attributes that have changed.

Creating Applications with CocoBase Transparent Persistence

Once Links and Maps are properly created, go to **File->Generate Java Code** in the main CocoAdmin Dialog to generate Java classes. The 'Default CocoNavigate Java Object' code generation target will generate a pure java class with no CocoBase interfaces.

After classes have been generated and compiled, CocoBase runtime classes can be used to persist instances of these classes. First, open a CocoBase connection as follows:

```
CocoDriverInterface myBase = CocoDriver.getCocoDriver(
    "thought.CocoBase.CocoPowder",
    "org.hsql.jdbcDriver",
    "jdbc:HypersonicSQL:hsql://localhost;cocoprop=cocofactory=CocoProxyFactory",
    "sa", ""
)
if(myBase.connect() == -1) {
    System.out.println("Failed connect!");
    System.exit(1);
}
...
```

Then create a CocoBase transaction object to manage any objects that are retrieved. Notice how the transaction object is configured through parameters and property settings.

```
thought.CocoBase.Transaction cocoTxn =
    new thought.CocoBase.Transaction(myBase, false);
Properties cocoTransProps = new Properties();
cocoTransProps.put("preserveCommit","true");
cocoTransProps.put("commitconnection","true");
cocoTransProps.put("throwExceptions","true");
cocoTransProps.put("updateOnlyChangedColumns","true");
cocoTxn.setProperties(cocoTransProps);

// Begin a new transaction.
cocoTxn.begin();
...
```

Then open a CocoBase Navigator object with the Navigation model to be used and also we register the Transaction object with the Navigation model:

```
// Instantiate a Navigator with the Link model created from
// the UML/XMI document
thought.CocoBase.navapi.Navigator navigator =
    new thought.CocoBase.navapi.Navigator (myBase, "company");
// Assign the current transaction to the Navigator
navigator.setTransaction(cocoTxn);
```

Now select the top level node to work with. Notice our use of a CocoProxyM class which 'wrappers' the pure java object model class and gives the class the compatibility with CocoBase through java reflection instead of requiring any special interfaces in the java class itself:

```
// Setup our object to query
Department dept = new Department();
department.setName("SALES");

// This will read & bind all 'Department' objects to the transaction.
Vector deptVector = myBase.selectAll(
                        new thought.CocoBase.CocoProxyM(dept),"Department");
```

Now it is possible to step through each of these objects and tell the retrieved object to be navigated through the `loadAllLinks` method which does the automatic navigation for that object:

```
for(int i=0; i< deptVector.size(); i++) {
    Department d = (Department)deptVector.elementAt(i);
    // Because the cascadeLoad flag is set to true in the direction
    // Department->Employees, the employees link will load automatically
    d = navigator.loadAllLinks(d,"Department");
    ...
```

```
    Vector emps = d.getEmployees();
    for (int j=0; j<emps.size(); j++) {
       // raise salaries by 20%
       emp.setSalary(emp.getSalary()*1.2);
       ...
    }
    ...

    // Once changes are made to an object graph those changes can be
    // synchronized using the updateAllLinks method such as:

    navigator.updateAllLinks(d,"Department",true);
}
```

You can then commit the buffered changes with:

```
cocoTxn.commit();
```

These code introductions are quite small, and can be done entirely server side with Entity or Session beans in J2EE environments with no model or object model intrusion. And for local non-j2ee apps the application intrusion is incredibly small with only the 'load', 'synchronize' and 'persist' operations requiring a single method call for each root node.

The Navigator supports one-to-one, one-to-many & many-to-many link definitions with cycle detection. It also detects this locally or by reconciling serialized or copied objects without object model or bytecode intrusion. This is truly transparent persistence that's architected and designed for the VM oriented Java language. There are also more advanced applications included in the demos/pguide/navapi subdirectory that demonstrate all of the mapping one-to-one, one-to-many & many-to-many relationships as well as an EJB using the Navigator system to manage a graph of java objects.

Conclusion

This Appendix is an Example describing how to add CocoBase Transparent Persistence to Object Models and applications from SQL database tables (graphic dialogs have been omitted for simplicity, and the options and dialogs may vary somewhat with new releases). This can be done in a local non-distributed environment. A wrapper can be used to wrapper objects and the CBSession class objects (Transparent Persistence mechanism). It is only a brief overview of what can be done with CocoBase. The architecture and implementation presented here is uniquely suited to work in every application from the tiny local application to the enterprise J2EE and to do so with superior performance and manageability.

I claim:

1. A system for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising:

a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, wherein the set of definitions is stored in a repository;

b) a set of definitions for the relationships between objects for an object language application, wherein the set of definitions is part of an object application navigation model;

c) a list of objects, or a set of objects that are to be persisted, wherein the list of objects or set of objects is part of an object application navigation model;

d) an object or set of objects as part of a computer implemented method that contains the computer logic capable of persisting an indicated object or set of objects;

e) an input method to provide the computer logic of d) with data corresponding to the location of information related to a), b) and c), and

f) at least one data source to in which persisted data may be stored,

wherein said system further comprises a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method providing a user the with the ability and method to query against an object graph, a complex data object graph or a portion thereof using an object query facility or syntax.

2. A system according to claim 1, wherein the object query facility utilizes EJB-QL query syntax or logically similar query syntax.

3. A system according to claim 2, wherein the object query facility utilizes the EJB-QL query syntax.

4. A system according to claim 1, wherein the object query facility utilizes a query syntax similar to a EJB-QL query syntax and including expanded functions not provided by EJB-QL querying.

5. A system according to claim 1, that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model or to permit object querying thereof.

6. A system according to claim 5, that provides persistence for a complex data object graph model on a distributed network environment.

7. A system according to claim 5, that provides persistence for a complex data object graph model on a distributed network environment, further comprising a generic ejb stateful session bean with logic for the computer implemented persistent methods or a similar complex object component with logic for the computer implemented persistence methods.

8. A system according to claim 1, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, further comprising a generic ejb stateful session bean with logic for the computer implemented persistent methods or a similar complex object component with logic the computer implemented persistence methods.

9. A system according to claim 1, that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model, further comprising a generic ejb stateful session bean with logic for the computer implemented persistent methods or a similar complex object component with logic for the computer implemented persistence methods.

10. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method to create or maintain transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising:

i) logic for a computer implemented method for creating a set of definitions for the relationships between a data source schema and objects capable of storing data for art object language application, wherein the logic for the computer implemented method is capable of causing the storage of the set of definitions in a repository;

ii) logic for a computer implemented method for creating and accessing a set of definitions for the relationships between objects for an object language application, wherein the logic for the computer implemented method is capable of causing storage of the set of definitions in a navigation model;

iii) logic for a computer implemented method providing an input means for a member selected from an object name, a list of objects, or a set of objects that are to be persisted, wherein the logic for a computer implemented method is capable of causing the storage of the input as part of a navigation model that can be accessed by ii);

iv) logic for a computer implemented method containing the logic and methods to persist an object or set of objects based upon access to (a) a set of definitions for the relationships between a data source schema and objects capable of storing data for an object language application, (b) a set of definitions for the relationships between objects for an object language application, and (c) an object name, a list of objects, or a set of objects that are to be persisted, and

v) logic for a computer implemented method that includes an input method which is capable of and has a method for informing iv) of the location of information relating to iv)(a), iv)(b) and iv)(c),

wherein said computer implemented method further comprises a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method to provide a user the with the ability and method to query against an object graph, a complex data object graph or a portion thereof using an object query facility or syntax.

11. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method according to claim 10, for creating or maintaining transparent persistence of a member selected from the group consisting of a data object, an object graph model and a portion of an object

graph model when a user of the system is creating, maintaining, accessing or navigating complex data objects as a complex data object graph model, comprising software code for generating, accessing, or maintaining a generic ejb stateful session bean with logic for the computer implemented persistent methods or for generating, accessing or maintaining a similar complex object component with logic for the computer implemented persistence methods.

12. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for providing a computer implemented method according to claim **10**, that does not require any modifications to an object model or the inclusion of any persistence byte code in the object model in order to provide persistence for all or a portion of the complex data object graph model, further comprising a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method of generating, accessing, or maintaining a generic ejb stateful session bean with logic for the computer implemented persistent methods or for generating, accessing or maintaining a similar complex object component with logic for the computer implemented persistence methods.

13. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **10**, that provides persistence for a complex data object graph model on a distributed network environment.

14. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **10**, that includes a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented method to automatically generate the logic for at least one of the routines from a relational data source schema, or to generate the logic from an object to relational mapping resource, said compiled or programming language software logic object code routine or set of object code routines that are stored in a permanent medium or in the memory of a computer system for a computer implemented persistence method comprising a computer implemented ability and method to query against the object graph or a portion thereof using a object query facility or syntax.

15. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **10**, that includes a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented object querying method, wherein the object query facility utilizes EJB-QL query syntax or logically similar query syntax.

16. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **15**, that includes a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented object querying method, wherein the object query facility utilizes the EJB-QL query syntax.

17. A compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory providing for a computer implemented method according to claim **15**, that includes a compiled or programming language software logic object code routine or set of object code routines stored in a permanent medium or in a computer system memory for a computer implemented object querying method, wherein the object query facility utilizes the EJB-QL query syntax and further and further utilizes logic and syntax for expanded querying functions that are not provided by EJB-QL querying.

* * * * *