TRANSPARENT EJB SUPPORT AND HORIZONTAL DATA PARTITIONING

COPYRIGHT NOTICE

[0001] A portion of the disclosure of this patent document contains material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the Patent and Trademark Office patent file or records, but otherwise reserves all copyright rights whatsoever.

BACKGROUND OF THE INVENTION

[0002] The present invention relates to application development using a common object framework (COF). More particularly, aspects of the invention relate to adapting the COF to be substantially compliant with a standard-based server-side component model and remote method invocation protocol. Additional aspects of the invention relate to horizontal partitioning of databases at a middle, business application tier, above a database tier. Combinations including staging, change history, and internationalization of fields within objects are further aspects of this invention. Particular aspects of the present invention are described in the claims, specification and drawings.

[0003] Multi-tiered applications have become a popular approach to deploying business logic in a distributed, reliable and scalable manner. In a multi-tiered system, a system services tier provides data base, transaction and other infrastructure. A middle tier provides access to business logic. A client tier accesses the business logic. The client tier may be users with thin or sophisticated client, or it may be programs such as web service clients. Multi-tiered applications are particularly useful to support multiple customers from a single application platform. Multi-tiered applications can be adapted to handle similarly formatted data for multiple customers, with advantages including code reuse and scaling.

[0004] J2EE and Enterprise JavaBeans™ (EJB) have emerged as an architecture for encapsulating and serving up business logic, integrated with mechanisms for data management, in a multi-tiered environment. An application server provides access to system services and infrastructure. The application server manages storage or persistence of objects that encapsulate both business logic and data. *See, generally*, Sun, *Simplified Guide to the Java™ 2 Platform, Enterprise Edition* (1999). Microsoft's rival architecture is known as .NET Framework and

DCOM. Monson-Haefel, Richard, *Enterprise JavaBeans 3d Ed.*, p. 17 O'Reilly (2001). An older framework with some similar characteristics is known as CORBA.

[0005] The life cycle of a multi-tiered, encapsulated business application includes both initial development and subsequent maintenance. (In this context, business application is used to include commercial and non-commercial applications that include one or more middle-tiers where so-called business logic resides.) Initial development of a multi-tiered, encapsulated business application may begin in a visual object-oriented analysis and modeling tool, progress through an object to relational database mapping process and be deployed as a multi-tiered, encapsulated business application. Maintenance may include staging and deployment of updated business application components. Deployment may require internationalization (abbreviated as I18N: "i" and "n" plus 18 letters in between) of dynamic data, such as product names and descriptions.

[0006] Tools that are used for initial development include visual analysis and modeling tools that conform to unified modeling language (UML) and produce an object-oriented model of the application and data that it will use. Rational Rose is a leading visual analysis and modeling tool. Other products include IBM's VisualAge, Borland's Enterprise Studio and NeuVis' NeuArchitect. Drafting tools also may be used for visual analysis and modeling, such as Visio or Flowcharter.

[0007] OR tools also may be used for development, to map an object-oriented model to a relational database. TopLink has been a leading tool for mapping an object-oriented model to a relational database. Rival tools are produced by Sun Microsystems, IBM, Borland, Oracle, Microsoft, Thought, Inc., and Tibco. Modeling tools that include some OR mapping capability generally are very limited and not well-adapted to supporting the range of available databases.

[0008] Application servers for multi-tiered, encapsulated business applications may be compliant with the J2EE standard or a similar standard. Application servers for J2EE and EJB 2.0 are provided by Sun, JBoss.org, Borland, IBM, BEA Systems, Warp Solutions, Macromedia, Oracle and many others. Application servers can rely on various databases for persistence, including databases provided by Oracle, Microsoft, IBM, Siebel, Sybase and others.

[0009] Two weaknesses of the available tools are limited integration and lack of support for features useful in maintenance and deployment. The limited integration means that users are required to reenter data or follow detailed programming guidelines so that applications developed in visual analysis and modeling tools and mapped in OR tools will comply with one of the rival standards for a multi-tiered, encapsulated business application deployment. The

features that would be useful to support, so that developers are not required to manually implement them using existing tools, include staging, history of deployment and internationalization of dynamic data. Staging may include testing new modules against live data and deploying tested modules. History refers to recording the deployment and optionally recording the staging and/or testing of new modules. Internationalization of dynamic data is support for adding additional fields and tables to support foreign language translations of dynamic data, by specifying the languages or countries to be added to one or more base languages or countries.

[0010] A desirable feature, not apparent in current systems, is horizontal partitioning of data at the middle tier, so that data can be located in multiple physical databases that are independent of each other and even databases built on different vendors' platforms, such as an Oracle database and SQL Server 2000. Here and below, the version "2000" is used only for reference and is not meant to suggest a limitation on versions of software that may be used to implement aspects of the present invention. Existing mechanisms for horizontal partitioning, supplied by database vendors, use multiple physical volumes within a single tightly managed system, all under control of the vendor's database manager. For examples of horizontal partitioning technology, see Klein, U.S. Pat. No. 6,453,313, *Database Management System and Method for Dequeuing Rows Published to a Database Table* (1992); Lohman et al., U.S. Pat. No. 6,112,198, *Optimization of Data Repartitioning During Parallel Query Optimization* (2000); Baru et al., U.S. Pat. No. 5,970,495, *Method and Apparatus for Achieving Uniform Data Distribution in a Parallel Database System* (1999). As Klien explains, transparent horizontal partitioning of large database tables is used to scale and distribute computational loads across the nodes and devices in a cluster. Horizontal partitioning means that a table is divided or partitioned into two or more files, with each partition storing the records (tuples) having a corresponding range of key values. For example, each of two or more partitions of a customer table might be used for records corresponding to customer names starting with corresponding range of letters A-L. Tables can also be partitioned based on a hash value. For example, using a hash function of a record index can have any of N (e.g., 4) values, a table can be divided into N partitions, with the hash function being used to determine the partition in which each table row is stored. Further, each partition may be stored on a different node or physical device of the system to facilitate distribution of the computational load on the system.

[0011] Integration of horizontal partitioning into the middle-tier, instead of the database manager, would be useful for supporting multiple customers with the same application.

Transparent horizontal partitioning support would free developers of revising their business logic as more customers are added or a database is repartitioned.

[0012] Accordingly, an opportunity arises for better support and integration of development at various stages from visual analysis and modeling through deployment and maintenance of multi-tiered, encapsulated business applications.

SUMMARY OF THE INVENTION

[0013] The present invention relates to application development using a common object framework (COF). More particularly, aspects of the invention relate to adapting the COF to be substantially compliant with a standard-based server-side component model and remote method invocation protocol. Additional aspects of the invention relate to horizontal partitioning of databases at a middle, business application tier, above a database tier. Combinations including staging, change history, and internationalization of fields within objects are further aspects of this invention. Particular aspects of the present invention are described in the claims, specification and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] Figure 1 is a high level block diagram of a multi-tiered application.

[0015] Figure 2 illustrates how a common object framework may be implementing using EJB on a J2EE server, providing a middle tier.

[0016] Figure 3 depicts objects and relationships for a simple shopping cart application visually modeled.

[0017] Figure 4 illustrates a CBOF tab 401 added into a pop-up window for specification of class properties.

[0018] Figure 5 illustrates a CBOF tab 401 added into a pop-up window for specification of operation properties.

[0019] Figure 6 depicts a CBOF model that can be adapted to EJB wrapping.

[0020] Figure 7 is a class diagram illustrating aspects of the classes CmoBase 701, CmsBase 702 and CmoAttribute 703, as adapted to support EJB and J2EE.

[0021] Adaptations of the COF tools and model to EJB and J2EE support are illustrated by figures 8-10.

[0022] Figure 11 generally illustrates horizontal partitioning.

[0023] Figure 12 illustrates classes that can be used to implement horizontal partitioning at the middle tier in a TopLink environment.

DETAILED DESCRIPTION

[0024] The following detailed description is made with reference to the figures. Preferred embodiments are described to illustrate the present invention, not to limit its scope, which is defined by the claims. Those of ordinary skill in the art will recognize a variety of equivalent variations on the description that follows.

[0025] Figure 1 illustrates the J2EE and EJB standards sponsored by Sun, which may be used in one embodiment of the present invention. *See*, Bodoff et al., *The J2EE Tutorial* , p. 10 Addison Wesley (2002). Sun explains the blocks in figure 1 as follows. The J2EE server 120 is the runtime portion of a J2EE product. A J2EE server provides EJB 125 and Web 121 containers and manages access to the backend database layer 130. An Enterprise JavaBeans (EJB) container 125 manages the execution of enterprise beans for J2EE applications. Enterprise beans 126, 127 and their container 125 run on the J2EE server 120. A web container 121 manages the execution of JSP page 123 and servlet 122 components for J2EE applications. Web components 122, 123 and their container 121 run on the J2EE server 120. An application client container 113 manages the execution of application client 112 components. Application clients 112 and their container 113 run on the client 110. An applet container manages the execution of applets. It includes a web browser 111 and one or more Java Plug-ins running on the client 110.

[0026] More generally, figure 1 illustrates a standard-based server-side component model and remote method invocation protocol. Microsoft .NET Framework and CORBA include similar components. Monson-Haefel, *Enterprise JavaBeans*, pp. 5-18. Aspects of the present invention are considered compliant with a standard if they comply with any version or off-shoot of the standard at or subsequent to a version identified herein or current at the time of application submission. Substantial compliance is useful, without requiring full compliance. It is recognized that full compliance with a standard will sometimes require implementation of features of a standard that considered unnecessary, too expensive or in flux. In addition, it is recognized that some implementations will include proprietary features that may be introduced to enhance a product and which may become non-standard as a standard evolves and customers insist on legacy support. Standard is meant to cover both committee controlled specifications, such as EJB, as well as vendor-controlled specifications, such as .NET Framework.

[0027] Figure 2 shows how a common object framework may be implementing using EJB on a J2EE server, providing a middle tier. Many of the components of figure 2 match the components of figure 1 and are numbered accordingly. In figure 2, the contents of the web container and one of the enterprise beans are described. In the web container 121, the applications include a common object framework (COF) 240A, a role application 241, a web services engine 242 and a collaborative process manager 243. The COF provides support for and access to persisted objects, reducing the need for the developer of the role application 241 to handle persistence, transaction management and related tasks. The COF optionally may support staging, change history, internationalization and horizontal partitioning, applying methods described in this application and the application incorporated by reference. The web services engine 242 and collaborative process manager are components used by Commerce One to provide web services; the details of these applications within the web container are not important to the present invention. Nonetheless, the solid arrow connecting the collaborative process manager 243 to the COF 240A and the dotted lines connecting the CPM 243 to the COF 240B and to the EJB 127 illustrate the option of substituting services and data support provided by the EJB container 124 for services available from COF 240A code that in the web container 121. Similar labeling of COF code in the web container 121 and an enterprise bean 126 as 240A, 240B is meant to indicate that COF code can be wrapped for EJB handling and with EJB-compliant interfaces with little or no rewriting of the COF code. As illustrated, COF can be made available in either the web container or an EJB container. Applications can access COF components in both containers. Applications can have active sessions simultaneously or concurrently with both sets of COF components. While figure 2 illustrates this implementation in the context of an EJB environment, one of skill in the art will recognize that the same approach can be applied to another standard-based server-side component model and remote method invocation protocol, such as DCOM.

[0028] The so-called common object framework provides a base implementation and tools to manage the full software life cycle of business objects, that is more fully described in the incorporated application. This includes the design, UML model, complete code & database generation that include transparent internationalization, staging, change history, and custom attributes. Tools are provided for ongoing maintenance of code, data, and schema that is database independent. The common object framework components fall into categories, including utilities library, business object framework and system management tools. The utilities library component includes common code level utilities like Exceptions

(internationalized), Logging, Cryptography, and Configuration access. It also includes internationalization functions such as Resource Manager, Date, Numeric, and Text. The business object framework is used for designing business objects, generating code for persistence, internationalization, data staging, version history, dynamic custom attributes, and full create/read/update/delete (CRUD) functionality. Included in the framework is support for extensibility of any business object, cache synchronization across multiple servers, and a scheduler. System management tools are based on a business object framework. They include data management tools to create databases, load data, upgrade database schema, and upgrade release changes in data. There are also extractors and installers for internationalization and localization kits.

[0029] The EJB-compliant COF or standards compliant COF may be implemented by and interact with a variety of software components. The design of the object model and generation of code or meta-data (machine readable data) may be implemented with Rational Rose 2001a or higher. Again, here and below, the version "2001a" is used only for reference and is not meant to suggest a limitation on versions of software that may be used to implement aspects of the present invention. When using Rational Rose, an add-in is installed to provide functionality illustrated in figures 4-5 and explained below. A database may be mapped, created and accessed using TopLink 3.6.1, SQL Server 2000, Oracle 8.1.7 or DB2, and an appropriate JDBC driver such as I-net OPTA 4.15, Seropto or IDS JDBC Lite 3.5.4. EJB containers such as WebSphere and WebLogic 7.0 may be used.

[0030] New and enhanced modules of an EJB-complaint COF may be combined in many useful ways. These modules generate remote and local EJB interfaces so that any COF object can be accessed and used as an EJB. They provide access to EJB-QL as the query language to find objects, using TOPLink 4.0 as the OR mapping tool. They manage database server sessions including support for simultaneous access to multiple databases and horizontal partitioning of data. Relationships across databases cannot be supported. They provide a client session for application access to any object. They provide a configurable re-try mechanism for lost database connections. They support integration with EJB JTA-based transactions and unique concatenated object ids that are unique across database servers, to preserve object ids across database migrations and to implement horizontal partitioning. Object change event notification are generated as configurable events for object creation, update, and delete. Callbacks are provided to the implemented object for post object creation, pre object update, post object refresh, and pre object delete. Design and code generation tools are provided for Rational Rose.

A database creation tool generates databases or DDL scripts based on a visually modeled design. Tools generate application server deployment descriptors for generated EJBs. Support for WebLogic 6.1sp2, and WebSphere 5.0 (needs EJB 2.0 spec support).

[0031] Figure 3 depicts objects and relationships for a simple shopping cart application visually modeled. The model in this figure was created using the Uniform Modeling Language (UML) notation for a class diagram. In this example, a CmsCart object 301 is used to provide a number of exposed methods. Use of a CmsCart instead of a CmoCart object implies that the object exists during a session and is not persisted for a subsequent session, in contrast to customers and orders, which are persisted. A cart 301 may be related by an order 311 to a CmoOrder object 303. One order object, in this example, relates to a shopping cart. A CmoOrder object 303 should, when completed, relate to a customer 313 via a CmoCustomer object 302. Along the opposite direction of relationship, a persistent customer 302 may relate 312 to more than one order 303. An order 303 contains one or more CmoProductLineItem objects 305. The relationship items 314 connects an order 303 to its line items 305. A line item 305 is filled in with products 304 that are connected to the line item by the product 315 relationship.

[0032] In UML, an object such as a shopping cart may be modeled as a class. By default, all top-level (i.e. no parent class) persistent objects in one embodiment of the invention inherit from a base object called CmoBase, as explained below in the context of Figure 5. Alternatively, transient objects may inherit from an object called CmsBase. The base object may be provided in a CAT (or unit) file that can be loaded into object modeling program, and referenced by other classes. The purpose of the base class is to provide basic functionality for all objects through a common interface and implementation.

[0033] Modeling a class in UML format with an object modeling program may include the following steps. First, a package hierarchy is created to specify the directory structure of where the class should exist and where it will be generated. Then the class name, class documentation, and class properties may be specified. Class properties supported include the following: Persistent, Transient, Abstract, Generate Code, Database Table Name, Is Stageable, Max Num of Versions, Table-Level database constraints (such as composite unique constraint), Is Displayed in GUI, Display Name, Display Description, Generate, and GenerateEJB, using a tool such as depicted in figure 4 and explained below.

[0034] Next, the attributes of a class may be specified. The attribute's name, type, documentation, database column name, database column length, null values allowed, and

whether or not the attribute is unique can be specified. In addition, the attribute can be static, final, and/or transient. Attributes may be defined protected so that subclasses and reflection code can access them. However, other classes and application code should only access them through getter and setter methods. The following attribute types are supported: int, boolean, float, long, short, double, char, String, Date, Time, Timestamp, Integer, Boolean, Float, Long, Short, Double, BigInteger, BigDecimal, Character, Object, byte, Byte, char, Character, EncryptedString, HashedString and TranslatableString.

[0035] These types are appropriately mapped to Java base types that may be mapped database-specific types. The Object, byte, and Byte may be mapped to BLOB-types in the database, to store streams of bytes. The char and Character may be mapped to CLOB-types in the database for storing large character streams. The EncryptedString and HashedString types are special types that are just a String but the code that is generated will encrypt/hash the value before setting the attribute and will decrypt the value when returning the attribute via the getter and setter methods. The TranslatableString attribute is also just a String but the code that is generated also generates an additional table with columns to represent translated values of that attribute. This allows particular attributes of a class to be specified to show that multiple translated values of that attribute can exist to support localization of object values, which are stored in the database.

[0036] Then, the relationships of a class may be specified. In UML, using a line with an arrow from one object to the other specifies a relationship or association. Each side of the association is called a role. The role name is equivalent to an attribute name on the class of the "from" side of the association. The role should be of protected control (just like attributes). Each role can specify a multiplicity. "0..1", "1", and "0..*" are supported. "0..1" specifies that at either zero or one instance of the object on that role side can be in the relationship. Similarly, "1" specifies that exactly one instance of the object on that role side should be in the relationship (typically signifying a not null constraint for the column that represents the association on the class on the from side of the relationship). Also, "0..*" specifies that 0 or more instances of an object can be in the relationship. This is equivalent to the relationship containing a collection of associated objects.

[0037] A relationship can also be bi-directional. This implies that each side of the association has visibility to the other side. Any permutation of the multiplicities specified above can exist in the bi-directional relationship. In addition, one side of the relationship can be shown as an aggregate (signified using an open diamond on the from side of the relationship). An

aggregate relationship specifies that the "from" class privately owns the instance(s) of the "to" class. So it is the responsibility of the "from" class to create and delete the instance(s) of the "to" class.

[0038] Object modeling program allows for additional properties to be defined that can be set on each individual package, class, attribute, relationship, and relationship role. A "CBOF" tab may be added to the properties for each type of object with object modeling program, as illustrated in Figure 3. This tab lists specific properties that can be set for each type of object. A module that generates meta data or machine readable data corresponding to the visual model can access any property of any object. Therefore, by adding properties, additional properties of the class can be set to customize subsequent generation of code and other support for that class.

[0039] In particular, the Rational Rose object modeling program allows extensions to be added that contain additional property values for various objects. When an object model is created (with features such as a Class, Attribute, Association, etc.), a set of properties can be specified and/or set on those object models. The set of properties can be extended and added to for customization purposes. Figure 4 illustrates a CBOF tab 401 added into a pop-up window for specification of class properties. A set or subset of properties can be selected 402 for manipulation. For model properties, a property name 411, value 412 and source 413 (such as default or override are displayed.) Controls all a user to manipulate the properties. Properties that have non-default values can be bold faced, highlighted or otherwise emphasized.

[0040] The following description lists properties that may be added for each type of object, and one or more manners in which it may be used to control generation of code and other support for a class:

[0041] 1. Project Properties: A single object model is considered a "project." Therefore, some properties can be set that are global to the entire corresponding project object.

- CopyrightNotice - override the copyright notice to be output in the generated code.
- ProjectName - override the class name of an OR mapping project class that will be generated.
- ProjectDescription - set a description that will be stored as a static variable in the OR mapping project class that is generated.

[0042] 2. Class Properties: A class may comprise a single UML class object created in accordance with a class diagram. By clicking on a set of standard properties, a new tab called "Cmo" may be added with properties including the following:

- Generate - set to "No" to avoid generating code for some classes that may exist solely for documentation purposes.
- GenerateEJB - set to "Yes" to generate one or more EJB interfaces to the class.
- TableName - override a default table name for a class. By default, the first 27 characters of the class name in upper case would be the table name.
- IsStageable - set to "False" to avoid generating any staging capability for this particular class, as further detailed belos.
- MaxNumVersions - set greater than 0 to set the number of past changes to keep on a particular class instance.
- ReplaceClass - specify the full path class name of an existing class that this class implementation replaces.
- WrapperPackageName - specify a package name of where a wrapper (ImplC) class is generated.
- ImplPackageName - specify a package name of where the persistent implementation and its primary wrapper (Impl and ImplW) classes are generated.
- StageablePackageName - specify the package name of where the persistent stageable implementation (ImplS) is generated.
- Constraint1 through Constraint7 - specify a table-level constraint on a class. For example, "UNIQUE (name, owningMember)"
- IsDisplayed - specify whether a class name should be displayed to a user in a given user interface.
- DisplayName - override the default display name of the class. By default, the class name is generated by removing any "Cmo" prefix and then adding spaces wherever there is a capital letter followed immediately by a lower-case letter.
- DisplayDescription - set a description that should be displayed for this class.

[0043] 3. Attribute Properties: An attribute can be added to a particular class for the base-type attributes described above. Attributes can have a new tab called "Cmo" with the following properties:

- ColumnName - override the default column name for this attribute in the table for the class of this attribute. By default, the first 30 characters of the attribute name in upper case with '_' to separate word boundaries would be the column name.
- I18NConstraintName - override the default constraint name for the _IN table that is added for this column (if the attribute is set to unique).

[0044] 4. Association Properties: An association can be added between two particular classes. Associations may include two sides, which are called roles. The association itself can have a new tab called "CBOF" with the following properties:

· RelationTableName - If the association represents a many to many or a unidirectional one to many relationship, then a middle or "relation" table may be added. Set this property to specify the relation table name. By default, the relation table names attempt to take the class names of both sides of the relationship and use "TO" in between. If this name is too large (more than 27 characters), then the class name initials (first letters of word boundaries) followed by a string hashcode of the class name are used.

[0045] 5. Role Properties: A role represents one side of an association. An association has Role side A and role side B. Therefore, if properties are added to the role object, two additional tabs may be added to association properties, namely, "CBOF A" and "CBOF B," with the following properties:

· ColumnName - override the default column name for this association in the table for the class of this association. By default, the first 30 characters of the role name in upper case with '_' to separate word boundaries would be the column name.

· GenerateForeignKeyConstraint - set to "False" if the foreign key constraint should not be generated for this side of the association.

· ForeignKeyConstraintName - override the default foreign key constraint name for this side of the association.

[0046] As discussed below, horizontal partitioning can be assigned as a project property, to include a group of objects, or as a class property, to include related objects. Once all the classes, attributes, and relationships have been modeled, then the code can be generated.

[0047] Operations also can be specified and properties of the operations assigned as illustrated in figure 5. The CBOF add-in tab 501 is depicted. At least two properties 521 are accessible, that indicate whether to generate an EJB local interface and/or an EJB remote interface. For each of the properties, a name 511, value 512 and source 513 are indicated. More generally, a local interface follows a first protocol with relatively little overhead, adapted to the calling and called objects residing in the same web container 121 or the same package. A remote interface follows a second protocol with greater overhead, adapted to remote method invocation. Remote method invocation protocols are provided for specifications and products such as CORBA, Java RMI, Microsoft Transaction Server, Microsoft COM/DCOM, Microsoft. NET

Framework and Enterprise JavaBeans (EJB). The choice of generating local and remote interfaces controls the calling overhead and/or exposure of an interface.

[0048] Figure 6 depicts a CBOF model that can be adapted to EJB wrapping. Referring now to FIG. 6, a block diagram of an object framework 610 is shown, in accordance with one embodiment the present invention. In alternate embodiments, object framework 610 may readily include various other components or configurations in addition to, or instead of, those components discussed in conjunction with the FIG. 6 embodiment.

[0049] Useful aspects of the COF/CBOF implementation may include combinations of: Use of Rational Rose for modeling objects with no restrictions on object model design. Code generation with placeholders or stubs for custom business logic methods. Generation of code to support run-time creation, read, update, and delete (CRUD) of objects without necessitating Rational Rose reverse engineering. Object to Relational (OR) mapping with little or no user interaction, beyond a user declaration of mapping parameters. Generation of persistence mapping information to persist the business object to a relational database. Definition and creation of relational tables and constraints, including information that is needed to create a database schema and tools that uses the information to create and/or migrate databases. Simple methods for Run-time creation, read, update, and delete (CRUD), accessible through a set of standard APIs to easily create, read, update, and delete any business object in the framework. Use of a transaction paradigm to create, update, and delete a set of objects using an atomic transaction in the database. Transactions can be long-lived because the database is not affected until commit time and optimistic locking is used. Data Staging to support changes to a business object without affecting the live production version of the object until the change is tested, approved, and deployed. Data change history support, including an audit trail of the base attribute changes of any object. Generation of the code and information to create a database schema that can be used for transparent storage and retrieval of translatable string attributes of a business object. Support for adding new attributes to any object dynamically at runtime without affecting the database schema. Support for object change notifications, including options for business objects to be configured to have an event generated upon creation, update, or deletion of the object. These events are persisted to the database and can be managed by an event notification service. Also, the business object itself can implement methods to perform some action after it is first constructed in memory (init()), after it is inserted into the database (postInsert()), after it is added to a transaction for update (postAddForUpdate()), before it is

updated in a transaction to the database (preCommit()), after it is loaded/refreshed from the database (postRefresh()), and finally, before it is deleted from the database (preDelete()).

[0050] In the FIG. 6 embodiment, a group of five base classes are shown above a horizontal axis 614, and may preferably include a base live object class, CmoBaseImpl 630, a base interface class, CmoBase 634, a base staging class, CmoBaseImplS 638, a base internationalization class, CmoBaseI18N 642, and a base wrapper class, CmoBaseImplW 644. The functionality and utilization of these base classes are generally discussed in the application incorporated by reference.

[0051] In the FIG. 6 embodiment, a group of root classes are shown between horizontal axis 614 and a horizontal axis 618, and may preferably include respective root classes that inherit from the foregoing corresponding base classes. In the FIG. 6 embodiment, the root classes may preferably include a root live object class, AbstractXImpl 646, a root interface class, AbstractX 650, a root staging class, AbstractXImplS 654, a root internationalization class, AbstractXI18N 658, a root wrapper class, AbstractXImplW 662, and a root custom class, AbstractXImplC 666.

[0052] In the FIG. 6 embodiment, a group of sub-classes are shown below horizontal axis 618, and may preferably include respective sub-classes that inherit from the foregoing corresponding base classes and root classes. In the FIG. 6 embodiment, the sub-classes may preferably include a sub-live object class, XImpl 670, a sub-interface class, X 674, a sub-staging class, XImplS 678, a sub-internationalization class, XI18N 682, a sub-wrapper class, XImplW 686, and a sub-custom class XImplC 690.

[0053] In addition, in the FIG. 6 diagram, a series of interface classes referred to above are shown to the left of a vertical axis 622. A series of wrapper classes are shown between vertical axis 622 and a vertical axis 626. In addition, a group of classes that are persistently stored in database 120 are shown to the right of vertical axis 626. Interfaces are the public Java interfaces that are exposed to users of the system. These interfaces define the get, set, add, remove, and custom business logic methods. The interfaces also contain the javadoc of APIs for reference. These interfaces are generated and should not be modified by hand, because handmade modifications will not survive a subsequent code regeneration.

[0054] In one embodiment, these classes interact with TopLink as a software component. Although some of the explanations that follow are specific to TopLink, one of skill in the art will appreciate that the same approach can be applied with other database mapping and management modules. The persistent classes are the implementation classes described to TOPLink as the classes containing persisted attributes and the mapping to the relational database. Three types of

persistent classes include Impl, ImplS, and I18N. The Impl class represents the exact live version of an object in the database. The ImplS class represents the shadow or staged version of a live object in the database. All get/set/add/remove methods declared in the interface class are implemented in these classes. All business logic APIs throw an UnsupportedOperationException. The I18N class contains the translatable string attributes and a locale for the translation. The Impl and ImplS classes have a one-to-many relationship to the I18N class. Whenever the get/set method of a translatable string attribute is called, then the appropriate instance of the I18N class is accessed based on the current locale set on the thread. These classes are generated and should not modified by hand.

[0055] The wrapper classes implement transaction support, staging, change history, and custom business logic functionality. There are two types of wrapper classes: ImplW and ImplC. The ImplW class is an abstract class that implements only the get/set/add/remove methods from the interface class. These methods appropriately check whether to retrieve or set a value on the Impl object, the ImplS object, or their appropriate TOPLink transaction clones. The set/add/remove methods throw a CeNotInTransactionException if the instance has not been added for update in a transaction. The get methods will throw a CeRuntimeException with a message of "Invalid wrapper accessed" if an attempt to access a cloned wrapper outside of a transaction is made. The ImplC class is the concrete class that is instantiated when any persistent object is accessed or created. This class inherits from the corresponding ImplW class. The ImplC class can be hand-modified to contain the implementation of any custom business logic method. The ImplW class is completely generated with each regeneration. The ImplC class typically is generated once and then checked into source control as an individual file that can be maintained. Note the inheritance structure of the ImplW and ImplC classes. The ImplW class inherits from the ImplC of its superclass. The ImplC inherits from its corresponding ImplW class. Also, note that ImplW does not implement the business logic APIs declared in the interface class. These are implemented by the ImplC class. A class can be declared abstract in the model and this would make the ImplC class abstract.

[0056] Also, note that persistent classes and associated TOPLink transaction clones have a reference to its corresponding wrapper. The persistent instances in TOPLink's cache point to the read-only versions of their wrappers. The clones point to the writable versions of their wrappers. When the clones are committed and merged back into the TOPLink's cached instances, then the corresponding wrappers become invalid. The clone's wrapper has a reference

to its corresponding read-only wrapper and this can be obtained by calling the getReadOnlyObject() method on the wrapper.

[0057] A base class hierarchy is needed to describe this set of files. FIG. 6 shows a picture of one embodiment of this base class hierarchy. At the top of the diagram, there are five base classes: CmoBaseImpl 630, CmoBase 634, CmoBaseImplW 644, CmoBaseImplS 638, and CmoBaseI18N 642.

[0058] CmoBase 634 is an interface class that all objects 332 will inherit from and implement. This interface contains the declaration of all of the basic attributes and operations that all objects 332 must implement. The various features of this object framework 610 contain attributes and methods that are added to this interface so that they can apply across all objects. CmoBase 634 also serves as a way to mark objects 332 that are generated by the mechanism provided in this framework. An alternative CmsBase class 702 may be provided, as shown if figure 7, for transient objects that do not require persistence.

[0059] CmoBaseImplW 644 represents the wrapper object that contains implementation to wrap access to the other implementation objects (CmoBaseImpl, CmoBaseImplS, and CmoBaseI18N). The CmoBaseImplW class implements the CmoBase interface. The purpose of the CmoBaseImplW class is to determine which instance of the Impl (for a live object) or ImplS (for a staged object) should be called to perform an operation. This class has four important relationships. The liveObject holds an instance of the CmoBaseImpl object, which represents the primary live business object in the database. The shadowObject holds an instance of the CmoBaseImplS object, which represents the shadow or staged version of the live business object in the database. Either the clonedObject holds a cloned copy of the liveObject or the shadowObject when the object has been put into a transaction for create or update. The liveClonedObject holds a cloned copy of the liveObject when the clonedObject contains a cloned copy of the shadowObject (which happens when the object is staged for update or create in a transaction).

[0060] liveObjects are given a read-only wrapper. A wrapper is made write-able by adding the read-only wrapper to a transaction, which returns a new wrapper including a clone of the liveObject that can be modified without affecting other users who are accessing the liveObject. When the transaction is committed, the clone is used to send updates to the database and to merge the changes into the liveObject. Once a write-able wrapper has been committed, it is no longer useful and the read-only wrapper must be accessed (this allows for the write-able

wrapper to be garbage collected to save on memory). An instance of this class is what is returned to the user when any action is taken (create, read, access relationship, and update).

[0061] CmoBaseImpl 630 represents the live persistent object in a database. This class contains properties that control the object-to-relational mapping. Such information allows creation of a schema including tables, columns, and constraints. This class contains the attributes defined in the object model including the attributes that represent relationships (either a single relationship with that object type or a many relationship using a collection type). This class also implements the CmoBase interface 634. The getters and setters for attributes and relationships are implemented to access the relevant attribute or relationship. The other business logic methods (declared in the model) throw UnsupportedOperationException to prevent accidental access to the method. Only the ImplW class should access this class and it should never call any of the business logic methods on the Impl class.

[0062] CmoBaseImplS 638 is essentially an exact copy of the CmoBaseImpl class but represents the shadow or "staged" values of the Impl instance. These staged values are stored in a separate table and are a separate instance, so a separate class is provided to represent that staged, which is mapped to the separate table. The utilization of CmoBaseImplS 638 are further discussed in the application incorporated by reference.

[0063] CmoBaseI18N 642 is the base class used for internationalization support. This class holds the base attributes that are needed for all tables holding translatable string values. When an object's Impl class is generated, an inner class that inherits from CmoBaseI18N 642 will also be generated if the class has translatable string attributes. Instances of CmoBaseI18N 642 correspond to language translations of the translatable string attributes of the Impl class. Additional instance are provided for staged values of the translations. The list of translations is kept at each level of a class hierarchy. This allows for adding/removing translatable string attributes at any time without requiring a hierarchy of CmoBaseI18N classes and tables. These features are described in more detail in the application incorporated by reference.

[0064] When an object is generated (for example AbstractX 646), the generated classes extend the appropriate base class. As shown in FIG. 6, AbstractX 646 is an interface that inherits from the CmoBase base interface. AbstractXImplW 662 inherits from CmoBaseImplW 644. AbstractXImpl 646 inherits from CmoBaseImpl 630. AbstractXImplS 654 inherits from CmoBaseImplS 638. Moreover, since there are translatable string attributes of AbstractX, AbstractXI18N 658 inherits from CmoBaseI18N 642.

[0065] There is also another custom class called AbstractXImplC 666 that inherits from AbstractXImplW 662. The ImplC classes contain the user-specific business logic code for the business logic methods that are added in the interface in the model. The ImplC class contains no other interface or implementation that is code generated. This allows the ImplC to be generated only once, and then only updated when business logic code needs to be added, changed, or removed. The other class types (Interface, ImplW, Impl, ImplS, I18N) are always re-generated from the object model 320 every time there is a change in the model. This solves the problem of generated code requiring any custom change by hand that needs to be reverse engineered into the model, and yet allows business logic implementation to be written that can change over time. The generated code is usable without requiring any code to be written in the ImplC. The ImplC provides a place for user-specific business logic methods to be implemented.

[0066] In addition, FIG. 6 shows how another class called X can further extend the AbstractX class. Each generated class type of X inherits from the appropriate class type of AbstractX. However, the XImplW 686 inherits from the AbstractXImplC 666, so that implementations of XImplC 686 can override any business logic method of AbstractX class just as if it had been a direct subclass. One aspect of the present invention is to automatically handle the complexities of the inheritance structure and the code to manage the relationships between these classes. Intuitively, the user only needs to worry about the visually modeled objects and business logic code implementation in ImplC.

[0067] Figure 7 is a class diagram illustrating aspects of the classes CmoBase 701, CmsBase 702 and CmoAttribute 703, as adapted to support EJB and J2EE. Persistent, user modifiable objects inherit from CmoBase 701. However, many attributes may be transient and not user modifiable, so their static values can be assigned to working copies of the objects created in memory. These transient attributes may inherit from CmsBase 702, which has properties and attributes adapted to transience. Examples of persistent, user modifiable objects at the attribute level include name, description, maximum number of versions, notifyAction. The CmoBase class 701 includes a primary key, "id" that is automatically generated. An ID is unique for a row within a DB and type. Thus, a completely and universally unique ID would be concatenate a database identifier, type and ID field. This concatenated unique identifier would satisfy the current EJB 2.0 requirement of a unique ID. A concatenated unique identifier also is useful for horizontal partitioning, discussed below, as ID in CmoBase 701 is only assured of being unique within a database and type. Additional attributes of CmoBase 701 include the illustrated status, versioning and staging attributes. As illustrated in figure 10, below, classes

CmoBaseLocal and CmoBaseRemote are constructed to inherit from or correspond to CmoBase. CmoAttribute 703 supports custom attributes for an object, as described further in the application incorporated by reference.

[0068] Adaptations of the COF tools and model to EJB and J2EE support are illustrated by figures 8-10. Preferably, business object interfaces in a system should extend from javax.ejb.EJBLocalObject, 1013 in figure 10. This pattern of extension provides access to invoke method calls on EJB objects running in the same Java virtual machine (JVM) without incurring the overhead associated with remote access. Each business object that needs to be accessed remotely requires a separate interface with the same name as the local interface and placed under "ejb" package. The remote interfaces extend javax.ejb.EJBObject 1012 and contain only methods that can be accessed remotely. In addition to the local and remote interfaces, ejb specification requires home interfaces to create and find ejb objects, as illustrated in figure 9. Each business object has a local home interface 912 which extends javax.ejb.EJBLocalHome 911 and a remote home interface 902 which extends javax.ejb.EJBHome 901.

[0069] The bean classes that contain code implementing business logic are usually named by suffixing "ImplC" to the interface name. For instance, in figure 8, PersonImplC 842 and EmployeeImplC 862 are objects intended to contain business logic corresponding to the Person and Employee interfaces 821, 841. Not all business methods in "ImplC" are remotely accessible. A remote method may be implemented having the parameters of the local method and a return value set to Serializable, as necessary to comply with EJB. To enable this, the framework generates a class with suffic "Bean" (e.g., PersonBean) where additional remote business methods can be implemented. In addition to providing the business logic implementation, a number of EJB-specific callbacks are implemented. These callbacks are provided for persistent objects by implementing javax.ejb.EntityBean interface 1011 in the CmoBaseImplW class (644 in figure 6). The callbacks implemented to provide EJB functionality include ejbCreate(), ejbPostCreate(), ejbFindByPrimaryKey(PrimaryKey key), ejbLoad(), ejbStore(), ejbRemove(), ejbActivate(), ejbPassivate(), setEntityContext(EntityContext ctx) and unsetEntityContext(). Among these callbacks, the ejbCreate() method is called when the client calls create() method on the home interface. The ejb container creates an instance of the bean (ImplC class) and calls ejbCreate() method on the bean. The implementation should provide methods to: acquire or create a CmoTransaction; add this bean to the transaction; initialize the wrapper object; preAllocate object Id; commit transaction; and Return PrimaryKey Object. This

method needs to be overloaded in many cases to pass in the not null database constraint parameters required to initialize the object.

[0070] The ejbPostCreate() method is called by the container immediately after ejbCreate() method call. This method should have the same number of arguments as ejbCreate() and provides additional initializations and allocate resources required by the during its lifespan.

[0071] The ejbFindByPrimaryKey(PrimaryKey key) method is called when the client invokes findByPrimaryKey() on the home interface. The implementation should provide methods to: indicate that no transaction is required; find the object in database based on object id (Use CmoClientSession.findById() method); and to return a PrimaryKey object. This method needs to be overloaded for custom queries.

[0072] The ejbLoad() callback method roughly corresponds to the 'read' functionality of entity bean. In one embodiment practicing aspects of the present invention, the bean is usually initialized during create() or find() method calls using TOPLink OR mapping and hence this method can be used for some post processing operations.

[0073] The ejbStore() callback method roughly corresponds to the 'update' functionality of entity bean. This functionality is already encapsulated in our CmoTransaction except the fact that a CmoTransaction need to be started and the object is added to the transaction for update.

[0074] The ejbRemove() callback method removes the object from the Data Store. A CmoTransaction is started and add the object for deletion.

[0075] During the activation process, the ejbActivate() method on the bean instance is called followed by the ejbLoad() method. Once the bean is activated and its state is properly synchronized to the underlying database record, the business method proceeds. Any subsequent calls to the business method will be directly delegated to the bean instance.

[0076] The ejbPassivate() is used during passivation. During the passivation process, the container invokes ejbStore() method on the bean instance. The bean instance is responsible for updating the database. After the database update, the container invokes the ejbPassivate() method on the bean instance giving itself an opportunity to free up any previously allocated resources before the bean is passivated.

[0077] The setEntityContext(EntityContext ctx) method is called with the context of the bean during bean initialization. The context is saved for future usage. The unsetEntityContext() complements the set context. The container may garbage-collect a pooled bean if it decided to free the resources used by the bean and in this case a transition back to the does not exist state

take place. During this transition period, the container calls this method. This method de-allocates any resources used by this bean.

[0078] Figure 8 illustrates application of an EJB-compliant CBOF object model to simple objects for person(s) and employee(s), with both local and remote access and interfaces. The person 821, 824 and employee 841, 834 objects implement from EJBLocalObject 811 and EJBObject 814. Implementation of the local object supports local, reduced overhead access. Implementation of EJBObject supports remote object access. Standard properties of the local objects are inherited from CmoBase 812, as indicated above. Throughout figure 8, the suffixes ImplW and ImplC are used for wrappers and concrete implementations. The EntityBean 813 is implemented by the CmoBaseImplW 822. Implementations of person 821 and employee 841 objects inherit from CmoBaseImplW 822. These implementations include PersonImplW 832, PrsonImplC 842, EmployeeImplW 852 and EmployeeImplC 862. As discussed in the context of figure 9, two home objects EJBLocalHome 853 and EJBHome 833 are provided for local and remote interfaces. Implementations of these home objects include PersonLocalHome 863, PersonHome 843, EmployeeLocalHome 864 and EmployeeHome 844.

[0079] Figure 9 generalizes part of figure 8 and illustrates compliance with the EJB convention requiring a home for objects. Homes are provided for both remote and local objects and interfaces, 901 and 911 respectively. Each business object has a remote home interface 902 which extends javax.ejb.EJBHome 901 and a local home interface 912 which extends javax.ejb.EJBLocalHome 911. The remote and local home interfaces 902, 912 are further extended by CmoApplicationConfigRemoteHome 903 and CmoApplicationConfigLocalHome 913.

[0080] Figure 10 also generalizes part of figure 8 and illustrates additional mechanisms to support EJB. The objects javax.ejb.EJBObject 1012 and javax.ejb.EJBLocalObject 1013 are extended to CmoBaseRemote 1022 and CmoBaseLocal 1023.

[0081] The EJB 2.0 specification supports relationships between EJB entitles. The three basic relationship cardinalities (one to one, one to many and many to many) are all supported. In addition, these relationships can be navigated using the new query language of EJB 2.0 specification. Relationships are managed thru abstract accessor methods. In the case of a relationship where the multiplicity of the associated object is many, the abstract accessor's parameter or return value type is a collection class representing the relationship. This collection class implements the java.util.Collection or java.util.Set interface. Optionally, the specification may add java.util.Map and java.util.List as return types. In the case of a relationship where the

multiplicity of the collection class in one, the abstract accessor's parameter or return value type is a single instance of the related object.

[0082] Relationship accessors in COF have parameter or return value types as an array of corresponding objects. To make them ejb accessible thru remote interfaces, additional methods for relationships should be provided in remote interfaces and implemented in ImplW. For instance, an Employee To Address Relationship may be one too many and bi-directional. Local and remote methods may be implemented that traverse a relationship, such as the Employee To Address Relationship example. For instance, local methods may include: public Address[] getAddresses(); public void setAddresses(Address[] address); public void addAddressElement(Address address); and public boolean removeAddressElement(Address address). Remote methods may include: public Collection getAddressesRemote(); public void setAddressesRemote(Collection address); public void addAddressElementRemote(Address address); and public boolean removeAddressElementRemote(Address address). The getAddressesRemote() method supports getting a list of objectId's for all the address objects associated with the Employee object. One can use the CmoQueryCursor mechanism for this since there is no need to cache any of the address objects. So, a new class called CmoObjectCursor which extends from CmoQueryCursor is created to support remote cursoring of EJB instances. This method supports finding or creating corresponding ejb objects using the remote home interface and storing them in a collection. It returns a collection of ejb objects. The setAddressesRemote(Collection address) method supports creating or finding corresponding ejb objects using local home interface. It stores the objects in an array. It calls a corresponding local method to update.

[0083] Transaction support is an important infrastructure service offered by the J2EE platform. The specification describes the Java Transaction API (JTA). JTA is an interface that provides access to the underlying transaction Manager. Some of the major interfaces include: javax.transaction.UserTransaction and javax.transaction.TransactionManager. The UserTransaction is exposed to application components, while the underlying interaction between the J2EE server and the JTA TransactionManager is transparent to the application components. The TransactionManager implementation supports the server's control of (container-demarcated) transaction boundaries. The JTA UserTransaction and JDBC's transactional support are both available to J2EE application components. The J2EE platform supports two transaction-management paradigms: declarative transaction demarcation and programmatic transaction demarcation. In a declarative demarcation, the container is responsible for starting, committing

and rolling back a transaction. The container starts a transaction for every operation on a bean by the client. The client has the ability to acquire the JTA transaction and perform several operations in one transaction. This will provide the flexibility of enforcing certain business logic that has dependency between operations. A transaction attribute supports declarative transaction demarcation and conveys to the container the intended transactional behavior of the associated EJB component's method.

[0084] Six transactional attributes are supported for container-managed transaction demarcation: Required, RequiresNew, NotSupported, Supports, Mandatory, and Never. A method with a Required transactional attribute is executed within a JTA transaction; depending on the circumstances, a new transaction context may or may not be created. If the calling component is already associated with a JTA transaction, the container will invoke the method in the context of said transaction. If no transaction is associated with the calling component, the container will automatically create a new transaction context and attempt to commit the transaction when the method completes.: A method with a RequiresNew transactional attribute is executed in the context of a new transaction. If the calling component is already associated with a transaction context, that transaction is suspended, a new transaction context is created, and the method is executed in the context of the new transaction, after whose completion the calling component's transaction is resumed. A method with a NotSupported transactional attribute is not intended to be part of a transaction. If the calling component is already associated with a transaction context, the container suspends that transaction, invokes the method unassociated with a transaction, and upon completion of the method, resumes the calling component's transaction. A method with a Supports transactional attribute supports the calling component's transactional situation. If the calling component does not have any transactional context, the container will execute the method as if its transaction attribute was NotSupported. If the calling component is already associated with a transactional context, the container will execute the method as if its transactional attribute was Required. A method with a Mandatory transactional attribute is called from the calling component's transaction context. Otherwise, the container will throw a javax.transaction.TransactionRequiredException. A method with a Never transactional attribute should never be called from a calling component's transaction context.

[0085] Programmatic transaction demarcation is the hard coding of transaction management within the application code. Programmatic transaction demarcation is a viable option for session EJBs, servlets, and JSP components. A programmatic transaction may be

either a JDBC or JTA transaction. For container-managed session EJBs, it is possible -- though not in the least recommended -- to mix JDBC and JTA transactions.

[0086] One aspect of the present invention is an ability to stage production data. To solve the problem of not doing replication across databases, one embodiment stores the staged values of the data in the same database as the live production values. However, such staged values should not exist in the same table as the live production values. If the staged data were in the same tables, then performance and size of tables would be affected. Therefore, for every object that needs staging capability, two tables may exist in the database. Since an additional table is then required, the OR mapping tool requires that another Java class represent that table. For Impl classes that are stageable, there are corresponding ImplS classes. Because the class structure and OR mapping are handled automatically, along with table schema creation, the effort required to add an additional table and/or class is substantially reduced.

[0087] To copy an existing object's attributes into a new object is generally straightforward. However, staging may also support changes in relationships between objects. To support all the various staging features, several rules may be defined:

[0088] 1. A live production object instance has a reference to its staged instance. An object can only be staged once at a time (in other words, it has a one-to-one bi-directional privately-owned relationship to its staged instance).

[0089] 2. Once an object is staged, its live instance cannot be modified. If the live instance could be modified, then merging of changes made in the staged values and live values would need to occur. This becomes unnecessarily complex. Typically, changes need to be staged or modified on the live object directly, not both.

[0090] 3. Relationships should be to live instances, not to a staged instance. Otherwise, many relationship mappings have to be managed (live to staged, live to live, staged to live, and staged to staged). In addition, when an object changes from staged to live or vise versa, then the relationships pointing to it need to manage that change. This rule simplifies these problems. Moreover, the live instance being referred to could be staged and has a reference to its staged instance, so in essence the various relationships can still be supported, but just not directly.

[0091] 4. Staging changes must be part of a stageable project. One way of managing a set of staging changes is to combine them in some container. A stageable project keeps track of all of the objects that have been staged. This project can then be deployed, deployed for test, or canceled.

[0092] 5. An object can only be staged once at a time and be part of a single stageable project at a time. Again, managing merges are generally not needed.

[0093] Supporting a one-to-one relationship is easy in the staged table due to foregoing rule #3. All relationships may refer to the live instance, so the staged table column that refers to another object just needs to refer to the live instance of that other object. A one-to-many relationship must use an intermediate relationship table even if the relationship is bi-directional. This prevents the live object table from having to know about any relationship to a staged instance.

[0094] The many-to-many relationship will also have an intermediate relationship table that links the staged instances to live instances. These intermediate relationship tables must be different from any existing live object relationship table. If the many-to-many relationship is bi-directional, then two intermediate relationship tables need to be created, one for each direction. This is because the relationship can be staged from either side of the relationship and has different values. The additional relationship tables are automatically and effectively generated.

[0095] An additional aspect of the present invention is horizontal partitioning at the middle-tier, as opposed to the so-called back office, database tier. Figure 11 generally illustrates horizontal partitioning. In this example, the object model includes components 1 (1101) to n (1102), accounts 1102 and BI 1104. Component 1 is mapped to Main 1111 database, which is a single database. Component n is mapped to both Main 1111 and Main 1112 databases, being horizontally partitioned across the databases. Component n may be a single object or a group of objects. A facility for grouping and specifying that the group of objects be horizontally portioned may be useful. Such a facility for setting project properties is described above and will be familiar to users of Rational Rose. Alternatively, a facility may be provided for assigning objects to one group or to a plurality of groups and then for setting group default properties or group-wide properties that may not be overridden. In either embodiment, it is useful to have a group of objects subject to horizontal partitioning (or not) as a result of a single setting. The mapping of components 1..n in this example may be configured by database declarations. A declarative approach such as a user completed table or a user-readable (e.g., XML) configuration file may be used to specify logical names of databases and/or to connect logical names of databases to objects and groups of objects. In this example, accounts and BI have been mapped, respectively, to the secure 1113 and BI 1114 databases.

[0096] The classes that support horizontal partitioning at the middle tier are adapted to the interface to the database or database infrastructure, to TopLink in this instance. Figure 12

illustrates classes that can be used to implement horizontal partitioning at the middle tier. This diagram is most easily understood beginning at the top left with CmoServerSession 1221. This object holds database connections and information required by the infrastructure. As the logical identification of the database is static, this object can have static data and static methods and can be shared by multiple users. A CmoSystemSession 1231 can be obtained from a server session 1221. Only a single system session object need be created for a server. The CmoClientSession object 1241 holds context and state information for each user of the server. It holds user specific information such as the user's locale and login. State information similar to the state information maintained for an HTTP session is maintained. The methods getDb and setDb are used to set context information. A client session may involve several databases. The relationship #dbClientSessions between a client session 1241 and any active database sessions 1242 is a 0..* relationship. No database client sessions are active when the client session is obtained. During the life of the client session, many database client sessions may become active.

[0097] The object CmoTransaction 1251 in this figure buffers data to be horizontally partitioned. The role of CmoTransaction is to maintain the status of a transaction, including long-lived transactions, until the transaction is read to commit. As the transaction is being assembled, information is collected that permits resolution of which database a new or modified object belongs in. When the transaction is ready to commit, it may be committed in one or more databases by one or more instances of CmoDatabaseTransaction 1252.

[0098] Horizontal partitioning support, including the middle column objects 1212, 1222, 1232, 1242 and 1252 can be added to an existing arrangement of infrastructure interfaces 1223, 1243 and 1253 plus sessions 1221, 1241 and 1251 that are exposed, without changing the manner in which the sessions are exposed or the operative business logic operative to which they are exposed. For instance, horizontal partitioning could be used to support two automobile manufacturers F and G using the same code and software. All data for F would be partitioned, for instance, into an Oracle database and all data for G would be partitioned into an SQL database. Once the context, F or G, was set, for example at login of a user, the partitioning of data would be transparent to the system. All calls for F's data would run against the Oracle database and all calls for G's data would run against the SQL database. The identical logic could be applied irrespective of whether the data service was provided to F or G. The code implementing the business logic could even be shared, because only the context setting would identify or distinguish data of F and G. Partitioning at this level would segregate the customers' data. Smaller databases might be handled more efficiently. A search by F would never look at

G's data, even at the back office, database server level. A mix-up in SQL, QL or other database inquiry code could not return data from more than one database, which reduces the chance that one customer could gain access to another customer's data. This would be particularly useful in competitive situations, such as when lawyers for plaintiff and defendant shared some components of a database but wanted to keep their work product strictly segregated or when competitors used the same value added service and the service vendor wanted to assure both customers (the competitors) that their data was secure. While the customer's data was kept secure from other customers, periodic maintenance functions could be accomplished by a system administrator iterating across multiple customer contexts.

[0099] The middle column objects 1212, 1222, 1232, 1242 and 1252 support access to multiple databases, based on context, where multiple databases store horizontally partitioned instances of the same objects. A CmoDatabaseServerSession object 1222 exists for each database active on a server. Databases are logically identified to CmoDatabaseServerSession 1222 by a logical name, which may be maintained as a keyed attribute dbName of the object. A getAllVersions method supports version management. This method can return the versions of objects in the database or a schema version. Code accessing the database or code granting access to the database should check that versions are not incompatible. The CmoClassConfiguration object 1212 connects objects to a database. One CmoClassConfiguration object 1212 represents an object accessible in a database. CuDatabaseConfig 1232 holds physical database configuration information corresponding to a logical database name. This information can be used to connect multiple physical databases to a single logical name, dependent on the context set. The CmoDatabseServerSession 1222 is connected by the #serverSession relationship to the infrastructure object ServerSession 1223. A CmoDatabaseClientSession 1242 manages the client session with the infrastructure object ClientSession 1243, taking into account the context of the client (F or G in our example above.)

[00100] Revisiting the operation of CmoTransaction 1251, when a transaction is ready to be committed, CmoTransaction invokes CmoDatabaseTransaction 1252 with the proper context set. CmoDatabaseTransaction 1252 connects with the infrastructure object 1253, for instance UnitOfWork in TopLink, to handle one or a two-phase commit of the object.

[00101] Two-phase commit is a transaction protocol that commits change across multiple physical databases in an atomic fashion. In one embodiment, software components to support two-phase commit include JTS-based Application Server (any J2EE 1.3 compliant App Server), distributed transaction support in JDBC driver (*e.g.*, Opta2000 and Seropto) and distributed

transaction support in database (*e.g.*, SQL Server and Oracle). COF and COF/EJB use TopLink to support transaction integration with a JTS-based (Java Transaction Service) transaction manager (typically provided by a J2EE application server.) Use of two-phase commit is an option. Two-phase commit is very expensive, so it is only used as necessary. User transactions that affect multiple objects and span multiple physical databases typically use a two-phase commit transaction. When an EJB object and a COF object need to be committed in a single transaction, a two-phase commit is appropriate. This protocol can be used with any combination of COF objects, horizontally partitioned objects, and EJB objects.

[00102] Note that horizontal partitioning does not depend on whether COF or another protocol is used directly or is encapsulated in EJB or another remote method invocation encapsulation. Middle tier encapsulation handles horizontal partitioning across independent databases, without relying on the database manager. This might be referred to as loosely coupled horizontal partitioning, as the independent databases can even be incompatible databases from different database vendors.

[00103] Again, relatively little business logic or system code needs to be changed to use horizontal partitioning. Some basic rules are listed here.

[00104] 1. If in the configuration file, only one database is being used, which means the value for property "database.projects" is single, then this database will be used always.

[00105] 2. If more than one database exists in the configuration file, then database information is resolved explicitly or implicitly when an operation is performed. Explicit resolution means "setDb(CuDatabaseConfig)" is called on the CmoClientSession object and/or CmoTransaction object. The CuDatabaseConfig objects typically are created during system startup time. "CmoServerSession. getAvailableDbObjects()" is used to get a list of available database objects for the whole system. For a particular class, "CmoServerSession.getDbObjectsForClass(Class aClass)" and "CmoServerSession.getDbObjectsForClass(String aClassName)" return a list of CuDatabaseConfig objects that the "class" resides on. Implicit resolution means that the CmoClientSession and/or CmoTransaction object can intelligently find which database is being worked on without calling "setDb(CuDatabaseConfig)" specifically in the application code, as explained below.

[00106] 3. When the database information cannot be resolved explicitly on CmoClientSession and CmoTransaction objects, the session objects will try to resolve this by using all the objects that got involved. If any object that got involved has some database

information (not null), this information will be compared to the database information that is associated with the session objects. If the database information for the session objects is null, then it will be set to be the database information from the object; else if the information is not the same, then an exception will be thrown. This logic will continue for all the objects that got involved. In order to assist identifying the database information for an operation and/or a transaction, COF introduced a new interface "CmoDbConflictResolver". A CmoDbConflictResolver object is configured for one project. This can be done in the configuration where the key is: database.conflictresolver.{full class name of project} = {full class name that implements CmoDbConflictResolver interface}.

[00107] 4. The resolver is called at the last possible moment. Meaning if an object is created, it won't be called until the object is attempted to be committed. This means, the resolver can expect that all the attributes of the object have been set for creation and can use them to determine which database the object should belong.

[00108] For the find case, the expression or attribute query, or attribute map query is passed so that the resolver can determine based on the attributes that are being queried which database to query against. We can never support query across databases in a single query. Again this is only needed if setDb has not already been called on the clientSession and the object being queried has been configured to exist in more than one database.

[00109] The way that CmoDbConflictResolver can be implemented is to use some configuration information or maybe some other dynamic data that tells it the mapping of some object attribute values to a particular physical database. This should be kept as simple as possible. For example, use the organization id of the object to determine which database, and maybe the databases are named based on orgId, so that it is a simple mapping. These database names are the "logical" names that are determined at configuration/installation time to give a name to a database and then its configuration information as to where its located, etc. So the name we use is a logical name and represents the name of the database configuration to access and not the actual physical database name in the database system.

[00110] Four functions are available on the client session to control the database information. They are: setDb(CuDatabaseConfig); getDb(); overrideDb(CuDatabaseConfig); and resetDb(). The getDb() just returns the current database that is set on the Client Session object; the overrideDb(CuDatabaseConfig aNewdb) and resetDb() use a stack mechanism to keep track of the change of db info on the client session. The overrideDb(CuDatabaseConfig aNewDb) will push the current db to the stack and set the current db to be "aNewDb" that is passed in. The

resetDb() will set the current db to be the top element of the stack if it is not empty; or else, it will silently do nothing. The setDb(CuDatabaseConfig aNewDb2) will set the current db to be "aNewDb2" and clear the stack. So if the setDb() is called when the stack has some information, the stack will be just cleared. The system client session object, which is a special client session, uses these four functions per thread base. Hence the database information on one thread has nothing to do with that on another thread.

[00111] The overrideDb(CuDatabaseConfig aDbConfig) and resetDb() functions can NOT be called on CmoTransaction although it is a subclass of CmoClientSession.

[00112] The CmoDbConflictResolver APIs include: public CuDatabaseConfig resolveCreate(CmoDbResolverInfo aDbResoverInfo) throws CeDbNameConflictResolveException; public CuDatabaseConfig resolveFind(CmoDbResolverInfo aDbResoverInfo) throws CeDbNameConflictResolveException. The CmoDbResolverInfo APIs include: public CmoClientSession getClientSession(); public Class getMainClass(); public HashMap getAttValuePairs(); public CmoFindObject getFindObject(); and public CmoBase getNewlyCreatedObject().

[00113] When a class implementing the CmoDbConflictResolver interface is developed, getNewlyCreatedObject() in CmoDbResolverInfo may be used to assist on the implementation of resolverCreate(CmoDbResolverInfo); similarly, getAttValuePairs() and/or getFindObject() in CmoDbResolverInfo may be useful in the implementation of resolverFind(CmoDbResolverInfo).

[00114] From the preceding description, it will be apparent to those of skill in the art that a wide variety of systems and methods can be constructed from aspects and components of the present invention. One embodiment is One embodiment of the present invention includes a method of generating application development code. This method alternatively may be practiced as a system or device carrying out steps of the method. The system or device may include several embodiments, combining steps, aspects, options and alternative embodiments of the method. The method also may be practiced as a magnetic medium impressed with a program carrying out steps of the method. This program may include several embodiments combining steps, aspects, options and alternative embodiments of the method. The generation of development code is particularly applicable to a multi-tiered, encapsulated business application environment. The method is based on a model of objects and relationships and operations of an object-oriented application design. Modeling of these objects, relationships and operations may be carried out visually, using a graphic tool. Properties assigned to the objects, relationships

and/or operations may include a first property indicating whether to generate an interface to a modeled object. It may include a second property indicating whether to generate local and/or remote interfaces to the specified operation. From the model, machine-readable data corresponding to the model and its properties is generated. This may data may embodied in an internal model or an exported model. The model may be machine-readable only or also human readable. The machine-readable data is used to map the model to the database. In one embodiment, the database is a relational database. Application development code is generated using the mapping and the machine-readable data. The generated application code may include code to persist and access objects in the database. It may further include interfaces that code to persist and access the objects, consistent with the first property. It may further include interfaces to the specified operations, consistent with the second property. The interfaces to the code and the specified operations may be substantially compliant with a standard-based server-side component model and remote method invocation protocol. The application development code alternatively may be substantially compliant with Enterprise JavaBeans (EJB) standard 2.0 or later. The code to persist and access EJB objects may be substantially compliant with a bean managed persistence (BMP) protocol. Alternatively, the application development code may be substantially compliant with Microsoft's DCOM specification. In another aspect of the present invention, the object-oriented application design may be substantially compliant with unified modeling language (UML) standard.

[00115] One aspect of the application development code method includes generating method stubs were custom business logic is to be added. Practicing this aspect of the invention, code that is regenerated when the object-oriented model for the system is modified is segregated from code that contains custom business logic and should not be overwritten or regenerated when the object-oriented model is modified.

[00116] Another aspect of the present invention further includes a plurality of third properties of the objects and relationships, the third properties controlling the mapping step. These properties are used to map objects and relationships to an underlying database, such as a relational database. These third properties may include whether to support staging of deployment of the modeled object. They may include whether to record a change history of deployment of the modeled object. It further may include whether to dynamically array one or more fields within the modeled object to internationalize those fields. In conjunction with the aspects of the present invention that involve dynamically arraying fields, the code to persist and access modeled objects may resolve which element of the dynamic array to retrieve based on a

language specified. This language may be implicitly specified by data present in an object being handled or may be explicitly specified by language.

[00117] Yet another aspect of the present invention is that a user may specify what application code to generate and the method steps of generating machine-readable data, mapping and additionally generating, can proceed without need for further actions by the user.

[00118] Another embodiment of the present invention is a method of assembling transactions to be processed by at least one database manager that supports transaction processing. This method alternatively may be practiced as a system or device carrying out steps of the method. The system or device may include several embodiments, combining steps, aspects, options and alternative embodiments of the method. The method also may be practiced as a magnetic medium impressed with the program carrying out steps of the method. This method may include providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support. Additionally, providing a version of the object-oriented framework that is substantially compliant with a standard-based server-side component model and remote invocation protocol. Examples of a standard-based server-side component model and remote invocation protocol including Enterprise JavaBeans (EJB), distributed common object model (DCOM) and COBRA. The method of assembling transactions further may include accepting or buffering transaction-related activity directed to one or more databases. This accepting or buffering may take place concurrently through the object-oriented framework and the standard compliant version of the object-oriented framework. One aspect of this embodiment is that the accepting step may include buffering objects and, on direction to commit the transaction, the method further includes invoking the database manager to process the transaction. This aspects includes accepting transaction-related activity by a mechanism other than the database manager. This aspect of the present invention may be practiced with any of the standards referred to above, including EJB, DCOM and COBRA.

[00119] An additional embodiment of the present invention is a method of assembling transactions to be processed by at least one database manager that supports transaction processing. As above, this method also may be practiced as a method, device or magnetic medium including a program practicing the method. Alternative methods of this embodiment are close variations on the assembling method described above. One variation includes providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support. This method additionally includes providing a version of the object-oriented framework adapted to EJB standard-compliant handling and

invocation. It further includes accepting transaction-related activity directed to one or more databases concurrently through the object-oriented framework and the EJB-standard compliant version of the object-oriented framework. In two other variations of this embodiment, the object-oriented framework is adapted to compliance with the DCOM specification or to COBRA. In any of these variations, the accepting staff may include buffering objects, and, on direction to commit the transaction, invoking the database manager to process the transaction.

[00120] Another embodiment is a method persisting data processed by at least one database manager. As above, this method also may be practiced as a method, device or magnetic medium including a program practicing the method. This method persisting data includes providing in object-oriented framework of methods that access and persist data through the database manager. It further includes providing standard-compliant interfaces to the methods that access and persist data through the database manager. These standard-compliant interfaces may be compliant with EJB, DCOM or COBRA. The method further includes accepting object access and persistence requests directed to one or more databases concurrently through the object-oriented framework and the standard compliant interface. When the standard is EJB, object requests may be submitted to a Java transaction services engine by both the object-oriented framework and the EJB standard-compliant interfaces.

[00121] An aspect of the present invention which may apply to any of the foregoing embodiments, is that the object-oriented framework is adapted to persisting fine-grained objects to databases. The standard-compliant interfaces, including EJB standard-compliant interfaces, are adapted to persisting objects in the transaction to more than one database.

[00122] Yet another embodiment of the present invention is a method of assembling transactions to be processed by at least one database manager that supports transaction processing. As above, this method also may be practiced as a method, device or magnetic medium including a program practicing the method. This method includes providing an object-oriented framework of methods that access and persist data through the data manager's transaction processing support and that also carry out an additional function. The additional function may be staging of objects for testing and addition to a production system. It may be tracking a change history of an object in a production system. Or, it may be dynamically arraying fields within an object to internationalize the fields. These additional functions may be combined pairwise or altogether. The method of assembling transactions further includes additionally providing a version of the object-oriented framework substantially compliant with a standard-based server-side component model and remote method invocation protocol, the

version also including one or more of the additional functions identified above. Separate embodiments include each of the additional functions individually, combinations of pairs of the additional functions, and all three additional functions. The standard with which this method of assembling transactions complies may be EJB, DCOM or COBRA.

[00123] A different embodiment of the present invention is a method of generating a database and a multi-tier business application that is horizontally partitioned. As above, this method also may be practiced as a method, device or magnetic medium including a program practicing the method. This method includes modeling objects and relationships and specifying operations of an object-oriented application design, wherein properties of the object include whether to horizontally partition an object across multiple databases. This approach may be applied either to objects or to groups of objects. The groups of objects may be represented by a single logical name. The method further includes generating machine-readable data corresponding to the model, including the properties. This machine-readable data may be only machine-readable or may also be human readable, such as in XML specification. The method further includes mapping the model to a plurality of databases using the machine readable data and additionally generating application development code from the mapping and from the machine-readable data. The application development code includes code to persist and access the objects across the plurality of databases. The code to persist and access the objects is a part of the middle program tier above a database tier. A further aspect of the present invention is that the code to persist and access the objects may invoke a plurality of independent databases. These independent databases may be supplied by different database vendors or may operate independently, though supplied by a single database vendor. A further aspect of this embodiment may include declaring a plurality of databases across which the object is partitioned and declaring one or more fields to be used to partition an object or group of objects into particular databases.

[00124] Yet another embodiment of the present invention is a method of implementing horizontal partitioning of objects across multiple independent databases, in a multi-tier business application including at least a database tier and a middle, business application tier. As above, this method also may be practiced as a method, device or magnetic medium including a program practicing the method. This embodiment includes declaring one or more fields in at least one horizontally partitioned object that can be used to assign a particular object to a particular database. It includes establishing connections with a plurality of databases to which the horizontally partitioned object may belong, though connections need not be established with all

of the databases to which the horizontally partitioned object might belong. Either before or after establishing connections, the method includes buffering one or more objects to find, to create, or to find or create, including the horizontally partitioned object. Upon direction to proceed with the find or create, resolving the particular database to which the object belongs using the declared fields and invoking the particular database in the database tier to perform the find or create. A similar embodiment is a method of implementing horizontal partitioning of objects across multiple independent databases, including establishing connections with a plurality of databases to which at least one horizontally partitioned object may belong. Further, setting or identifying the particular database to which the horizontally partitioned object belongs. Before or after the setting step, buffering one or more objects to find, to create, or to find or create, including the horizontally partitioned object. Upon direction to proceed with the find or create, invoking the particular database in the database tier to perform the find or create. The plurality of databases to which the horizontally partitioned object may belong may be collectively referenced with a single logical name. Particular databases among the plurality of databases are logically mapped using one or more attributes of the horizontally partitioned object. These attributes may be fields of the horizontally partition object.

[00125] While the present invention is disclosed by reference to the preferred embodiments and examples detailed above, it is understood that these examples are intended in an illustrative rather than in a limiting sense. Computer-assisted processing is implicated in the described embodiments. Accordingly, the present invention may be embodied in methods for computer-assisted processing, systems including logic to implement the methods, media impressed with logic to carry out the methods, data streams impressed with logic to carry out the methods, or computer-accessible processing services. It is contemplated that modifications and combinations will readily occur to those skilled in the art, which modifications and combinations will be within the spirit of the invention and the scope of the following claims.

[00126] We claim as follows:

CLAIMS

1. A method of generating application development code for a multi-tiered, encapsulated business application, including:

visually modeling objects and relationships and specifying operations of an object-oriented application design, wherein properties of the objects, relationships and/or operations comprise:

a first property indicating whether to generate an interface to the modeled object; and

a second property indicating whether to generate local and/or remote interfaces to a specified operation;

generating machine readable data corresponding to the visual model, including the properties;

mapping the visual model to a database, using the machine readable data; and

additionally generating application development code from the mapping and the machine readable data, comprising:

code to persist and access the modeled objects in the database;

interfaces to the code to persist and access the objects, consistent with the first property; and

interfaces to the specified operations, consistent with the second property.

2. The method of claim 1, wherein the application development code is substantially compliant with an Enterprise JavaBeans (EJB) standard 2.0 or later.

3. The method of claim 2, wherein the code to persist and access the objects is substantially compliant with a Bean Managed Persistence (BMP) protocol.

4. The method of claim 1, wherein the application development code is substantially compliant with a .NET Framework specification.

5. The method of claim 1, wherein the object-oriented application design is substantially compliant with a unified modeling language (UML) standard.

6. The method of claim 1, wherein the properties further comprise a plurality of third properties of the objects and the relationships controlling the mapping step.

7. The method of claim 6, wherein the third properties controlling the mapping step include whether to support staging of deployment of the modeled object.

8. The method of claim 7, wherein the third properties controlling the mapping step further include whether to record a change history of the deployment of the modeled object.

9. The method of claim 1, wherein the properties of the objects further comprise one or more fourth properties indicating whether to dynamically array one or more fields within the modeled object to internationalize the fields.

10. The method of claim 9, wherein the code to persist and access the objects resolves which element of the dynamic array to retrieve based on a language specified.

11. The method of claim 1, wherein a user specifies what to generate and the steps of generating machine readable data, mapping, and additionally generating proceed without need for further actions by the user.

12. The method of claim 1, wherein the database is a relational database.

13. A method of assembling transactions to be processed by at least one database manager that supports transaction processing, including:

providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support;

additionally providing a version of the object-oriented framework substantially compliant with a standard-based server-side component model and remote method invocation protocol;

accepting transaction-related activity directed to one or more databases concurrently through the object-oriented framework and the standard compliant version of the object-oriented framework.

14. The method of claim 13, wherein the standard is Enterprise JavaBeans (EJB).

15. The method of claim 13, wherein the standard is Distributed Common Object Model (DCOM).

16. The method of claim 13, wherein the standard is CORBA.

17. The method of claim 14, wherein the accepting step includes buffering objects, further including, upon direction to commit the transaction, invoking the database manager to process the transaction.

18. The method of claim 15, wherein the accepting step includes buffering objects, further including, upon direction to commit the transaction, invoking the database manager to process the transaction.

19. The method of claim 16, wherein the accepting step includes buffering objects, further including, upon direction to commit the transaction, invoking the database manager to process the transaction.

20. A method of persisting data processed by at least one database manager, including:

providing an object-oriented framework of methods that access and persist data through the database manager;

providing EJB standard-compliant interfaces to the methods that access and persist data through the database manager; and

accepting object access and persistence requests directed to one or more databases concurrently through the object-oriented framework and the EJB-standard compliant interfaces.

21. The method of claim 20, wherein the object requests are submitted to a Java transaction services engine, irrespective of whether they are made through the object-oriented framework or the EJB standard-compliant interfaces.

22. The method of claim 20, wherein the object-oriented framework, without the EJB standard-compliant interfaces, is adapted to persisting fine-grained objects and the EJB standard-complaint interfaces are adapted to persisting objects in a transaction to more than one database.

23. A method of assembling transactions to be processed by at least one database manager that supports transaction processing, including:
providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support and that also stage objects for testing and addition to a production system; and
additionally providing a version of the object-oriented framework substantially compliant with a standard-based server-side component model and remote method invocation protocol, the version including the staging of objects.

24. The method of claim 23, wherein the standard is Enterprise JavaBeans (EJB).

25. The method of claim 23, wherein the standard is Distributed Common Object Model (DCOM).

26. The method of claim 23, wherein the standard is CORBA.

27. A method of assembling transactions to be processed by at least one database. Manager that supports transaction processing, including:

providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support and that also track a change history of an object in a production system; and

additionally providing a version of the object-oriented framework substantially compliant with a standard-based server-side component model and remote method invocation protocol, the version including the tracking of the change history.

28. The method of claim 27, wherein the standard is Enterprise JavaBeans (EJB).

29. The method of claim 27, wherein the standard is Distributed Common Object Model (DCOM).

30. The method of claim 27, wherein the standard is CORBA.

31. A method of assembling transactions to be processed by at least one database manager that supports transaction processing, including:

providing an object-oriented framework of methods that access and persist data through the database manager's transaction processing support and that also dynamically array fields within an object to internationalize the fields; and

additionally providing a version of the object-oriented framework substantially compliant with a standard-based server-side component model and remote method

invocation protocol, the version including the dynamic arraying of fields to internationalize the fields.

32. The method of claim 31, wherein the standard is Enterprise JavaBeans (EJB).

33. The method of claim 31, wherein the standard is Distributed Common Object Model (DCOM).

34. The method of claim 31, wherein the standard is CORBA.

35. A method of generating a database in a multi-tiered business application that is horizontally partitioned, including:

modeling objects and relationships and specifying operations of an object-oriented application design, wherein properties of the objects include whether to horizontally partition an object across multiple databases;

generating machine readable data corresponding to the model, including the properties mapping the model to a plurality of databases, using the machine readable data; and

additionally generating application development code from the mapping and the machine readable data, including code to persist and access the objects across the plurality of databases, wherein the code to persist and access the objects is a middle program tier above a database tier.

36. The method of claim 35, further including a declaring of a plurality of databases across which the object is partitioned and declaring one or more fields of the object used to partition the object into particular databases among the plurality of databases.

37. A method of implementing horizontal partitioning of objects across multiple independent databases, in a multi-tiered business application including at least a database tier and a middle, business application tier, including:

declaring one or more fields in at least one horizontally partitioned object that can be used to assign a particular object to a particular database;

establishing connections with a plurality of databases to which the horizontally partitioned object may belong;

buffering one or more objects to find or create, including the horizontally partitioned object; and

upon direction to proceed with the find or create, resolving the particular database to which the object belongs using the declared fields and invoking the particular database in the database tier to perform the find or create.

38. A method of implementing horizontal partitioning of objects across multiple independent databases, in a multi-tiered business application including at least a database tier and a middle, business application tier, including:

establishing connections with a plurality of databases to which at least one horizontally portioned object may belong;

setting the particular database to which the horizontally partitioned object belongs;

buffering one or more objects to find or create, before or after the setting step, including the horizontally partitioned object; and

upon direction to proceed with the find or create, invoking the particular database in the database tier to perform the find or create.

39. The method of claim 38, wherein the plurality of databases are collectively referenced with a single logical name and the particular databases among the plurality of databases are logically mapped to one or more additional attributes.

40. The method of claim 39, wherein the additional attributes are fields of the horizontally partitioned object.