(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 February 2002 (07.02.2002)

PCT

(10) International Publication Number
WO 02/10911 A2

(51) International Patent Classification[7]: G06F 9/00

(21) International Application Number: PCT/US01/41471

(22) International Filing Date: 31 July 2001 (31.07.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/629,156 31 July 2000 (31.07.2000) US

(71) Applicant: **COMPOZE SOFTWARE, INC.** [US/US]; 1554 Paoli Pike, #228, West Chester, PA 19380 (US).

(72) Inventors: **HEIST, Peter G,**; 333 Lancaster Avenue, Apt. 316, Frazer, PA 19355 (US). **SPOSETTI, Jeffrey T,**; 1618 Barker Circle, West Chester, PA 19380 (US).

(74) Agents: **GREENBAUM, Michael C,** et al.; Blank Rome Comisky & McCauley LLP, The Farragut Building, Suite 1000, 900 17th Street, NW, Washington, DC 20006 (US).

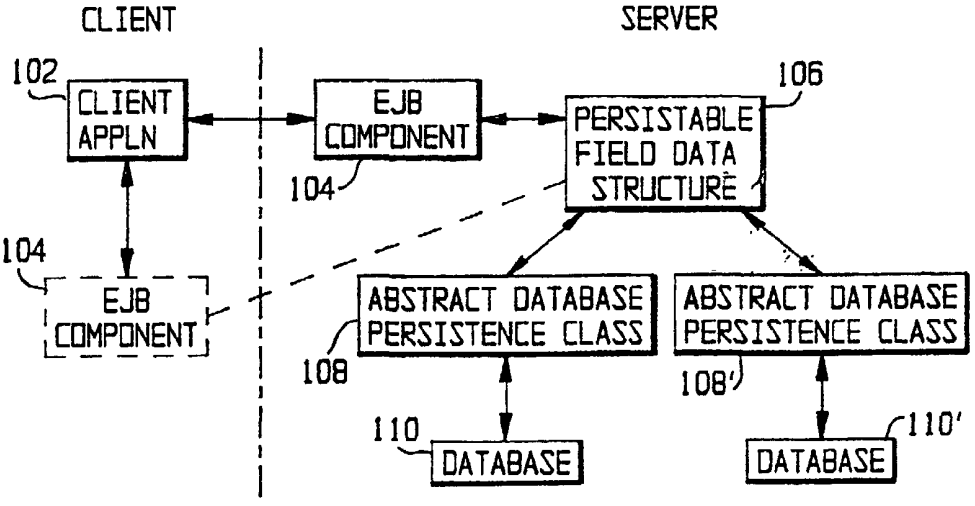
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: REUSABLE SOFTWARE COMPONENT WITH FLEXIBLE PERSISTENCE MECHANISM



(57) **Abstract:** In Java or another programming language, software components load data from a database and store data in the database. When a software component is initialized, that component's fields are queried to map those fields to fields in the database and to form a persistable field graph of the mapping. During that phase, commonly used SQL statements are generated. Operations between the software component and the database are handled by traversing the persistable field graph. Classes referenced by the software component can be similarly mapped; in that case, each field in the class is mapped to a field in the database. The needed SQL statements can be generated by an abstract database persistence class which translates database calls into the specific version of SQL supported by the database, so as to provide transparent support for different types of databases.

WO 02/10911 A2

REUSABLE SOFTWARE COMPONENT WITH FLEXIBLE PERSISTENCE MECHANISM

Field of the Invention

The present invention is directed to a reusable software component and in particular to a reusable software component with improved persistence for use in an application server.

Description of Related Art

A well-designed software component is a piece of software code which is reusable, flexible, and extensible. A software component conforms to a component standard so that it may be used by any application that supports that type of component. Different types of components include graphical controls, hardware drivers and server-side business logic, or middleware.

A component is *reusable* when it can be used in multiple scenarios. For example, a bank or other lending institution might be able to use the same account component for multiple applications. A component is *flexible* when its configuration can be changed to control the way it behaves in a particular environment. For example, the account component may contain a data field for storing a variable annual interest rate. A component is *extensible* when additional code may be written to augment or replace code in an existing component and extend its behavior to handle new scenarios.

A parallel can be drawn between a software component and a piece of hardware in a personal computer (PC). Many standards exist in the PC which allow parts with similar functions to be interchanged. For example, the PCI standard allows manufacturers of various video cards, sound cards and network interface cards to provide their hardware to different types of personal computers. Upgrades may be made to an individual component with minimal impact on the overall system. Likewise, an upgrade may be made to a software component without adversely affecting its users.

The term *persistence* describes the action of storing and loading the data and state of a

component to and from an external storage mechanism. That external storage mechanism may be a network, a file on disk, or a database. Databases are used to store, retrieve and efficiently search large amounts of data. When a database is used for component persistence, the fields (or individual pieces of data) of a component are typically mapped over to data types that the database understands. For example, an integer may be mapped over to a more general "number" in the database. That mapping often occurs at compile time or is configured using a tool that aids in the generation of persistence code.

Java is a programming language and platform created by Sun Microsystems for developing applications. Java offers several advantages over a native language such as C or C++ which allow runtime persistence. Some of the advantages include:

Virtual machine (VM) - isolates the programmer from platform specific issues so that programs written in Java can run on any hardware platform;

Garbage collection - managing memory is easier because the VM is responsible for freeing memory when it is no longer in use;

Introspection - the fields and methods in a Java object can be analyzed, invoked or modified at runtime, allowing efficient runtime switching of code behavior; and

Serialization - Java objects can be easily converted to and from an appropriate data format for transfer over a network or for storage in a database.

Those features (in particular introspection) open up an opportunity for runtime object examination and thus runtime persistence.

The Enterprise JavaBeans (EJB) specification describes how server-side business logic, or middleware components are specified in Java. EJB components come in two forms: entity beans and session beans. Entity beans have their data persisted to a database, whereas session beans are used mainly without persistence to provide additional business logic or to provide convenience methods that interact with and manipulate other entity beans.

Clients connect to EJB components to execute their business logic. Those clients can include a web-based application, a wireless device or a handheld computer. Application Servers provide the functionality (such as transaction support, database connectivity and component persistence) needed to run EJB components. Components are "deployed" on an application server for use by clients. That deployment process prepares an EJB interface to be executed in the application server environment.

In order to persist the fields of an entity bean, a number of methods must be implemented in each bean:

ejbCreate() – create a new bean instance by creating a new row in the database to store its fields;

ejbLoad() – update or initialize a bean instance in memory by loading its fields from the database;

ejbStore() – update the database by storing the fields of a bean instance to it;

ejbRemove() – remove a bean instance by deleting the corresponding row from the database;

setEntityContext() – set an object that is used to obtain the current configuration, transaction status and security settings;

unsetEntityContext() – invalidate the current entity context;

ejbPassivate() – release any resources not needed by the bean instance while it is not in use; and

ejbActivate() – re-initialize the resources being used by a bean instance so that it is ready for use.

Additionally, beans may implement "finder methods". Finder methods are responsible for locating bean instances stored in the database by using a mechanism such as SQL to search and retrieve the database fields.

EJB components have a primary key. A primary key is a field or multiple fields in the component that are guaranteed to be unique for each bean instance. The primary key may be a standard Java type or a custom class.

The EJB 1.1 specification is the most current EJB specification now available. In the EJB 1.1 specification, entity beans may be persisted in one of two ways:

Container-managed persistence - the application server, or container, is responsible for persisting the fields of the component over to the database fields. The way that fields in the component are mapped over to fields in the database is controlled by an XML "deployment descriptor", a file that is modified prior to deployment.

Bean-managed persistence - the component itself is responsible for persisting its fields to the database.

The following are traits of the EJB 1.1 specified container-managed and bean-managed persistence:

	Container-Managed Persistence	Bean-Managed Persistence
Component to database field mapping	In XML deployment descriptor, changed at deploy time	Directly in code, changed at compile time
Finder methods	In XML deployment descriptor, changed at deploy time by a container specific XML schema	Directly in code, changed at compile time
ejbCreate(), ejbLoad(), ejbStore(), and ejbRemove() methods	Implemented by EJB container or application server, behavior is controlled by the field mapping in the deployment descriptor at deploy time	Directly in code, changed at compile time

Serializable Java objects other than ones mapped in Java's SQL package	Persisted to database as a "blob" of binary data that the database is not able to search, or all dependent class fields must be specified in XML.	Persisted to database as a "blob" of binary data that the database is not able to search, but individual fields may be mapped by hand.

The above-noted traits of container- and bean-managed persistence yield advantages and disadvantages for each mechanism.

Container-managed persistence offers the following advantages: It provides a means to map bean fields over to database fields using XML; no SQL code is needed for ejbCreate(), ejbRemove(), ejbLoad(), and ejbStore(); and finders may be specified in the XML descriptor rather than in the code, so that changes to finders require only redeployment, not recompilation. It also has the following disadvantages: Specifying finders and other deployment options is application server specific, as is supplementing a bean with custom finders; persisting serializable classes into blob (binary large object) fields is not always possible; conditionally persisting large fields is application server specific (if it can't be done then large fields must be loaded and stored with every method call); changing fields during the development process requires a change to the deployment descriptor and a re-deployment.

Bean-managed persistence offers the following advantages: SQL is used to persist fields, allowing complicated mapping between components fields and database fields; and finders are flexible, as they may use any SQL code they wish; and there is an easy opportunity to write SQL code that is database specific. It also includes the following disadvantages: Complete SQL code must be written for every bean to perform the ejbCreate(), ejbLoad(), ejbStore(), and ejbRemove() methods, so that the beans become more cumbersome to develop and debug; and an average of 40% more lines of code are added over container-managed persistence.

Summary of the Invention

It will be readily apparent from the above that a need exists in the art to combine the advantages of container- and bean-managed persistence. It is therefore a primary object of the invention to simplify development for component persistence while retaining flexibility.

It is another object of the invention to reduce the overhead required for field and interface management.

It is still another object of the invention to perform persistence while maintaining database independence.

It is yet another object of the invention to combine quicker development and easier deployment of software components with a flexible persistence mechanism.

To achieve the above and other objects, the present invention is directed to various techniques for developing reusable software components. In Java or another programming language, software components load data from a database and store data in the database. When a software component is initialized, that component's fields are queried to map those fields to fields in the database and to form a persistable field graph of the mapping. During that phase, commonly used SQL statements are generated. Operations between the software component and the database are handled by traversing the persistable field graph. Classes referenced by the software component can be similarly mapped; in that case, each field in the class is mapped to a field in the database. The needed SQL statements can be generated by an abstract database persistence class which translates database calls into the specific version of SQL supported by the database, so as to provide transparent support for different types of databases.

The techniques are implemented using Java and Enterprise JavaBeans, although other technologies can be used instead or in addition. The Enterprise JavaBeans architecture is extended to allow quicker development of EJB components, easier deployment of components across platforms and a flexible persistence mechanism.

The following techniques are employed and may be used separately, although it is contemplated that they will be used together:

Automatic Bean-Managed Persistence (ABMP) - ABMP is a technology that simplifies development for EJB component persistence, while still retaining flexibility.

IPersistable - The IPersistable technique works in concert with ABMP. IPersistable seamlessly maps individual component class fields to the database thus reducing the required overhead of field and interface management.

Abstract Database Persistence (ADP) - ADP is a technology that abstracts database details away when performing persistence thus maintaining database independence. ADP allows deployment of beans across application servers.

Automatic Bean-Managed Persistence (ABMP) is the core technique presented herein. ABMP is a form of bean-managed persistence that uses Java's introspection facility to map component fields to database fields during runtime. The ability to initiate ABMP removes the excess code needed for bean-managed persistence while still retaining its innate flexibility.

Brief Description of the Drawings

A preferred embodiment of the present invention will be set forth in detail with reference to the drawings, in which:

Fig. 1 shows the interaction among the various components of the preferred embodiment;

Fig. 2 shows a flow chart of the initialization of a software component of the preferred embodiment;

Figs. 3A-3D and 4 show UML (unified markup language) diagrams of the initialization of Fig. 2; and

Fig. 5 shows a UML diagram of the implementation of database independence in the preferred embodiment.

Detailed Description of the Preferred Embodiment

A preferred embodiment of the present invention will be set forth in detail with reference to the drawings, in which like reference numerals refer to like elements throughout.

An overview of a system on which the preferred embodiment will be described with reference to Fig. 1. A client application 102 on a client side, which can be a Web browser, a Java applet or the like, communicates with an Enterprise JavaBeans component 104, which can run on either the client side or a server side. The component 104 includes data fields and code to process the data in the fields to perform useful work for the client application 102. The component 104 loads the contents of its fields from a database 110, and stores the contents of its fields in the database 110, by traversing a persistable field data structure 106 which maps each field of the component 104 to a component of the database 110. An abstract database persistence class 108 translates the database calls of the component 104 into SQL statements understood by the database 110. Thus, different implementations of SQL can be handled transparently to the component 104. Should the component 104 need to persist its data to another database 110′ having a different implementation of SQL, another abstract database persistence class 108′ can be used.

The operations performed by the server are shown in the flow chart of Fig. 2. In step 202, the component 104 is initialized. In step 204, the fields in the component 104 are queried using Java's introspection. A field in a component 104 can refer to a class with multiple fields, one of which may in turn refer to a class with multiple fields. To accommodate that possibility, the server checks in step 206 whether any field refers to a class. If so, the server goes to the fields in the class in step 208 and queries them in step 204. Steps 204-208 are performed until all of the fields in all of the classes are queried. Thus, an indefinite number of levels of fields can be accommodated.

Then, in step 210, the fields in the component 104, and all of the fields queried in steps

204-208, are mapped to fields in the database 110 (or 110'), and a persistable field graph is formed to represent that mapping and stored in a data structure. In step 212, commonly used SQL statements needed to implement persistence are generated. In step 214, the field graph is traversed to load data from the database to the component, to store data from the component to the database, or to update the datebase to reflect the creation or destruction of instances of the component.

The various techniques used to implement the functionality described above will now be described in further detail.

Automatic Bean-Managed Persistence (ABMP) is the core technique presented herein. ABMP is a form of bean-managed persistence that uses Java's introspection facility to map component fields to database fields during runtime. The ability to initiate ABMP removes the excess code needed for bean-managed persistence while still retaining its innate flexibility.

The ABMP algorithm consists of:

1. Persistable Field Graph – That is the data structure that stores a graph of all component fields that must be mapped to the database. Each node in the persistable field graph contains the bean's object field, the database field, the field type, and any child nodes associated with an IPersistable, which will be described in detail below. Only Java fields which are non-transient, non-static, and non-final are considered to be persistable.

2. Initialization Procedure – When an entity bean class using ABMP is first constructed, it creates the persistable field graph by recursively querying the component's fields using introspection. During that phase, commonly used SQL statements required for the bean's persistence methods are generated. That process of analyzing the fields and creating SQL statements only needs to be executed once per entity bean object, but may be done additional times as a bean is passivated and activated.

3. Default implementation of ejbCreate() – Traverses the persistable field graph to

set the bean object's fields and add a new row to the database.

4. Default implementation of ejbLoad() – Traverses the persistable field graph to set the bean object's fields from the values in the database.

5. Default implementation of ejbStore() – Traverses the persistable field graph to set the values in the database from the bean object's fields.

6. Default implementation of ejbRemove() – Removes the bean instance by removing its row from the database.

7. Default implementation of setEntityContext() – Store the entity context in a member variable accessible by the bean.

8. Default implementation of unsetEntityContext() – Set the stored entity context to null.

9. Default implementation of ejbPassivate() – Set fields to null and null out references to persistable field graph to allow the garbage collector to free the unused resources.

10. Default implementation of ejbActivate() – Restore bean fields and the persistable field graph so that the bean may be used again.

11. An implementation of findByPrimaryKey() – Execute a SQL statement to find a bean instance by its primary key.

12. An implementation of findByField() – Execute a SQL statement to find a list of bean instances with fields that are equal to a specified value. Since that is a common requirement, many finders will be able to use that method to avoid the need for repetitive SQL statements.

13. An implementation of findByFieldAndSort() - Execute a SQL statement to find a list of bean instances with fields that are equal to a specified value, and return those instances sorted by the same or another field.

14. The loadField() method – Load a single field and any dependent fields from the database. That may be used in combination with a transient field to defer the loading of a large

field from the database until it is required.

15. The storeField() method – Store a single field and any dependent fields to the database. That may be used in combination with a transient field to defer the storing of a large field to the database until it is required.

Advantages of ABMP over container-managed and bean-managed persistence:

- All required bean-managed persistence methods are already implemented. All that remains to create a bean is to add its fields and implement its business methods. No deployment descriptors need to be modified.
- Most finder methods are already implemented. If any additional finder logic is required it may be added with the same flexibility as bean-managed persistence.
- The loadField() and storeField() methods may be used to prevent the loading large fields until it is absolutely necessary. Without that feature, a one-megabyte field would need to be loaded and stored with the invocation of each business method.
- During the development process, adding a field to a bean only requires adding it to the bean class and the database schema. No XML or SQL changes are necessary as would be necessary with container-managed or bean-managed persistence.

Diagrams in unified modeling language (UML) of the operation of ABMP will be set forth in Figs. 3A-3D.

Fig. 3A shows the first step in the initialization procedure of the component 104 and corresponds to steps 202-212 of Fig. 2. APMP 300 identifies persistable fields in the component 104 using introspection and maps them to field in the database 110. A data structure maintains the mapping.

Fig. 3B shows the second step and corresponds to step 214 of Fig. 2. SQL statements are

generated for ejbLoad(), ejbStore(), ejbCreate(), ejbRemove(), loadField(), storeField() and finder methods.

Fig. 3C shows the ejbLoad() procedure, while Fig. 3D shows the ejbStore() procedure. In each of those procedures, as described above, the data to be loaded or stored traverse the persistable field database structure to set the fields in the component 104 from the corresponding fields in the database 110 or vice versa.

The IPersistable technique works in tandem with ABMP. In the same way ABMP allows a bean's fields to be mapped over to the database, the IPersistable technique allows the fields in any of the bean-referenced classes to be similarly mapped.

The database field names are determined by the field's position in the persistable field graph using a database specific separator character. For example, if the '$' character is used, a database field name might be beanField$referencedField. That means that in the class referenced by the bean field beanField there is a field named eferencedField. The nesting of referenced classes may be arbitrarily deep.

There is one additional field per IPersistable class that stores a oolean value stating whether or not the IPersistable is null. Using that value an entire reference to a large IPersistable class may be nulled by changing one field.

Advantages of using IPersistable vs. Java's serialization facility for persisting referenced classes from a component:

- Fields are stored in the database using their primitive database types rather than a blob containing the Java serialization format. That allows the database to search its fields with maximum efficiency.
- Adding fields or making other non-destructive changes to the persistable class's signature does not require setting the Java serialization ID in the class.
- Renaming and deleting fields does not require the addition of code to provide for a

serialization schema evolution.

Fig. 4 shows one field in the EJB component 104 mapping over to multiple fields in the database 110 through the IPersistable 400. The IPersistable technology is responsible for maintaining that mapping and ensures that all fields in the database and in the component are updated when that field is needed.

The following code defines the IPersistable field data structure:

```
/**
 *
 * This class stores info about a persistable field or object. For performance
 * reasons accessor methods are not used, but direct access to members is
 * forced.
 *
 * @author              PHeist (03/00)
 */
public class        PersistableField
{
/**
 * Java field for this node
 */
public Field                        i_field;
/**
 * Field name represented locally
 */
public String                       i_sFieldName;
/**
```

```
    * Field name in database
    */
    public String                   i_sDBFieldName;
    /**
    * Bean object for the field
    */
    public Object                   i_obj;
    /**
    * Mappable type
    */
    public int                      i_iType;
    /**
    * Parent persistable field
    */
    public PersistableField         i_parent;
    /**
    * Child persistable fields
    */
    public PersistableField[]       i_children;
    /**
    * List of children (only valid during construction)
    */
    public LinkedList               i_childrenList;
    /**
    * If true, this field is persistable
```

```
*/
public boolean          i_bPersistable;
/**
* Constructor.
*
* @param          field          the Java field
* @param          sFieldName     the Java field name
* @param          sDBFieldName   the database field name
* @param          obj            the Java object
* @param          iType          the mappable type (see MAPTYPE_ constants)
* @param          parent         the persistable field's parents
* @param          children       the persistable field's children
* @param          bPersistable   true if the field is persistable
*/
public          PersistableField
(
Field           field,
String          sFieldName,
String          sDBFieldName,
Object          obj,
int                     iType,
PersistableField   parent,
PersistableField[] children,
boolean                 bPersistable
)
```

```
{
i_field = field;
i_sFieldName = sFieldName;
i_sDBFieldName = sDBFieldName;
i_obj = obj;
i_iType = iType;
i_parent = parent;
i_children = children;
i_childrenList = new LinkedList ();
i_bPersistable = bPersistable;
// force accessibility so that private variables may be accessed
try
        {
        i_field.setAccessible (true);
        }
catch (SecurityException ex)
        {
        Debug.i().println ("AbstractAutoEntityBean: unable to force" +
        " field reflection, please grant permission " +
        "java.lang.reflect.ReflectPermission \"suppressAccessChecks\"");
if (!m_bPrintedSuppressAccessException)
        {
        Debug.i().printStackTrace (ex);
        m_bPrintedSuppressAccessException = true;
        }
```

```
        }
    }
} // PersistableField
```

Since most of the SQL code for the bean is centralized in the ABMP implementation class, an opportunity arises to standardize the way that database interactions occur. Although SQL is a standard language for interacting with databases and in theory any SQL statement should work on every database, in practice the various database vendors adhere to standards that differ slightly.

In the ABMP class, instead of executing SQL code directly, the Abstract Database Persistence (ADP) class is called. That occurs from ABMP's ejbCreate(), ejbLoad(), ejbStore(), ejbRemove(), and finder method implementations. The actual implementation of the ADP class in use may vary and is controlled at runtime.

Advantages of the ADP mechanism:

- Multiple databases may be supported without having complicated SQL code in the bean.
- The database being used may be switched at runtime without making changes to ABMP or any of the beans that have avoided using custom SQL statements.
- Adding support for a new database consists of only writing a new ADP class.
- EJB code becomes mostly isolated from the SQL differences in each database.

Fig. 5 shows a UML diagram of the relationship between ADP and ABMP. As shown in Fig. 5, a component 104 communicates with ABMP 300, which includes certain database calls which vary among SQL implementations and therefore use ADP 500. ADP 500 thus serves as an interface between ABMP 300 and the databases 110 and 110′.

While a preferred embodiment of the present invention has been set forth in detail, those skilled in the art who have reviewed the present application will readily appreciate that other embodiments can be realized within the scope of the invention. For example, while the preferred

embodiment uses Java, Enterprise JavaBeans and SQL, other technologies can be used. Also, while a client and a server are shown in Fig. 1, any distribution of processing between the client and the server can be implemented; for that matter, the invention can be used on a stand-alone computer. Therefore, the present invention should be construed as limited only by the appended claims.

We claim:

1. A method of implementing a persistable software component on a computing device, the computing device also implementing a database or being in communication with a server implementing a database, the method comprising:

(a) initializing the software component;

(b) analyzing the software component to identify persistable fields in the software component;

(c) mapping the persistable fields identified in step (b) with fields in the database to form a persistable field graph; and

(d) storing in the computing device a data structure representing the persistable field graph.

2. The method of claim 1, further comprising (e) setting the persistable fields in the software component from the fields in the database by traversing the persistable field graph.

3. The method of claim 2, further comprising (f) setting the fields in the database from the persistable fields in the software component by traversing the persistable field graph.

4. The method of claim 3, further comprising:

(g) creating and destroying instances of the software component; and

(h) adding and removing rows in the database in accordance with step (g) by traversing the persistable field graph.

5. The method of claim 4, wherein at least one of steps (e), (f) and (h) comprises forming a statement to access the database.

6. The method of claim 5, further comprising (i) providing an abstract database persistence class for each of a plurality of types of databases, the abstract database persistence class converting database calls from the software component to statements for a corresponding type of database; and wherein the statement to access the database is formed by the abstract

database persistence class.

7. The method of claim 6, wherein:

the software component accesses a plurality of databases; and

step (i) comprises providing a plurality of abstract data persistence classes, one for each o of the plurality of databases.

8. The method of claim 1, wherein the persistable field graph also maps fields in classes referenced by the software component to the fields in the database.

9. The method of claim 8, wherein the data structure representing the persistable field graph comprises a Boolean field for each of the classes, wherein an entire one of the classes can be nulled in the persistable field graph by setting the Boolean field for that class to zero.

10. A computing device implementing a persistable software component, the computing device also implementing a database or being in communication with a server implementing a database, the computing device comprising:

processing means for (a) initializing the software component; (b) analyzing the software component to identify persistable fields in the software component; and (c) mapping the persistable fields identified in step (b) with fields in the database to form a persistable field graph; and

storage means for storing a data structure representing the persistable field graph.

11. The computing device of claim 10, wherein the processing means also sets the persistable fields in the software component from the fields in the database by traversing the persistable field graph.

12. The computing device of claim 11, wherein the processing means also sets the fields in the database from the persistable fields in the software component by traversing the persistable field graph.

13. The computing device of claim 12, wherein the processing means also creates and

destroys instances of the software component and adds and removes rows in the database by traversing the persistable field graph.

14. The computing device of claim 13, wherein the processing means also forms a statement to access the database.

15. The computing device of claim 14, wherein the processing means provides an abstract database persistence class for each of a plurality of types of databases, the abstract database persistence class converting database calls from the software component to statements for a corresponding type of database; and wherein the statement to access the database is formed by the abstract database persistence class.

16. The computing device of claim 15, wherein:

the software component accesses a plurality of databases; and

the processing means provides a plurality of abstract data persistence classes, one for each o of the plurality of databases.

17. The computing device of claim 10, wherein the persistable field graph also maps fields in classes referenced by the software component to the fields in the database.

18. The computing device of claim 17, wherein the data structure representing the persistable field graph comprises a Boolean field for each of the classes, wherein an entire one of the classes can be nulled in the persistable field graph by setting the Boolean field for that class to zero.

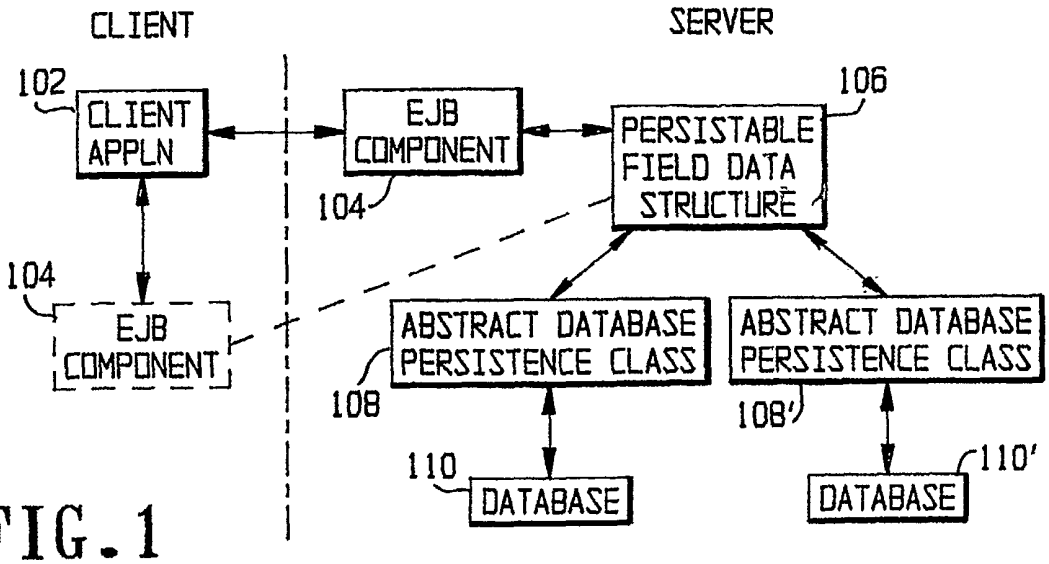

CLIENT
SERVER
102
CLIENT APPLN
EJB COMPONENT
104
106
PERSISTABLE FIELD DATA STRUCTURE
104
EJB COMPONENT
ABSTRACT DATABASE PERSISTENCE CLASS
108
ABSTRACT DATABASE PERSISTENCE CLASS
108'
110
DATABASE
110'
DATABASE

FIG. 1

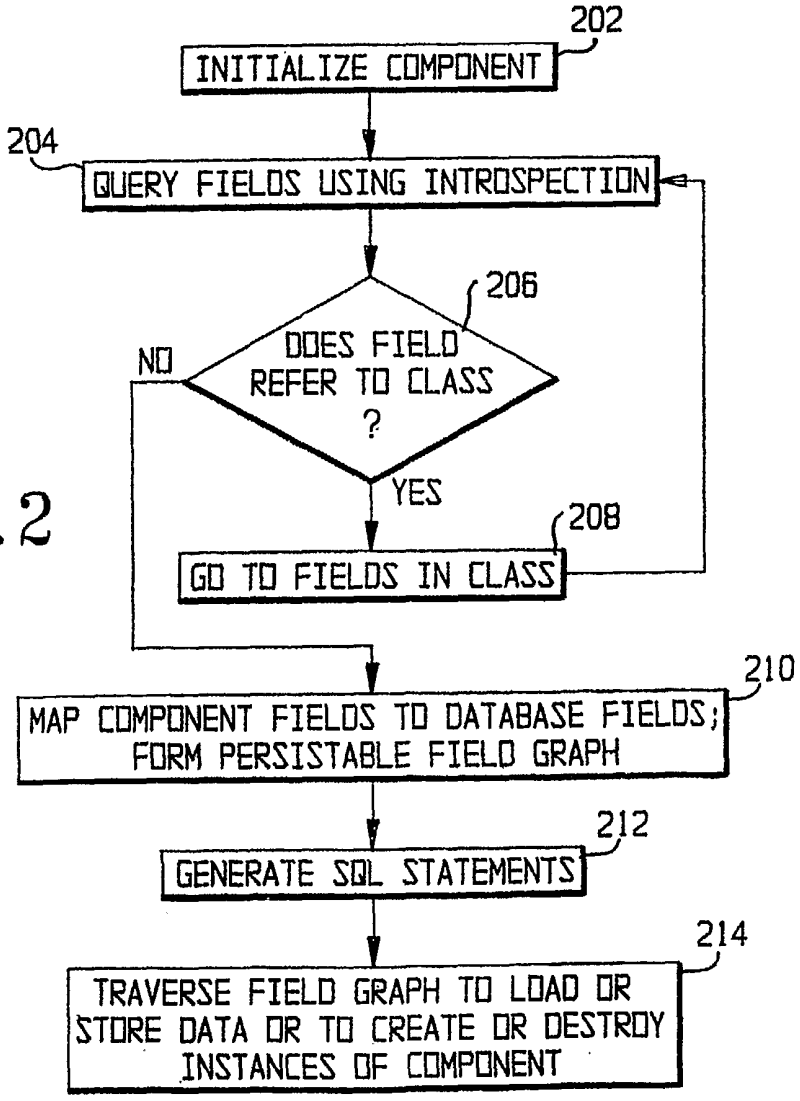

202
INITIALIZE COMPONENT
204
QUERY FIELDS USING INTROSPECTION
206
DOES FIELD REFER TO CLASS ?
NO
YES
208
GO TO FIELDS IN CLASS
210
MAP COMPONENT FIELDS TO DATABASE FIELDS; FORM PERSISTABLE FIELD GRAPH
212
GENERATE SQL STATEMENTS
214
TRAVERSE FIELD GRAPH TO LOAD OR STORE DATA OR TO CREATE OR DESTROY INSTANCES OF COMPONENT

FIG. 2

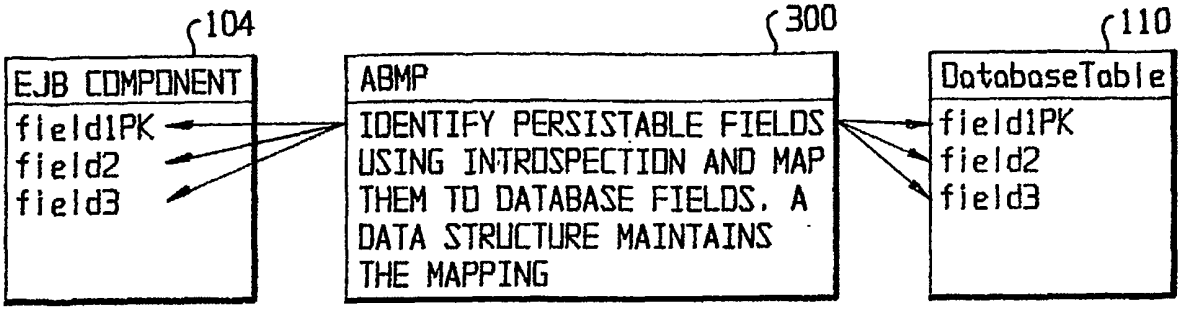


FIG. 3A

ABMP
-Generate SQL for ejbLoad( ) - INSERT INTO DatabaseTable (field1PK,field2,field3) VALUES (?, ?, ?) -Generate SQL for ejbStore( ) - SELECT field2,field3 FROM DatabaseTable WHERE field1PK=? -Generate additional SQL statements for ejbCreate ( ), ejbRemove( ), loadField( ), storeField( ), and finder methods.

FIG. 3B

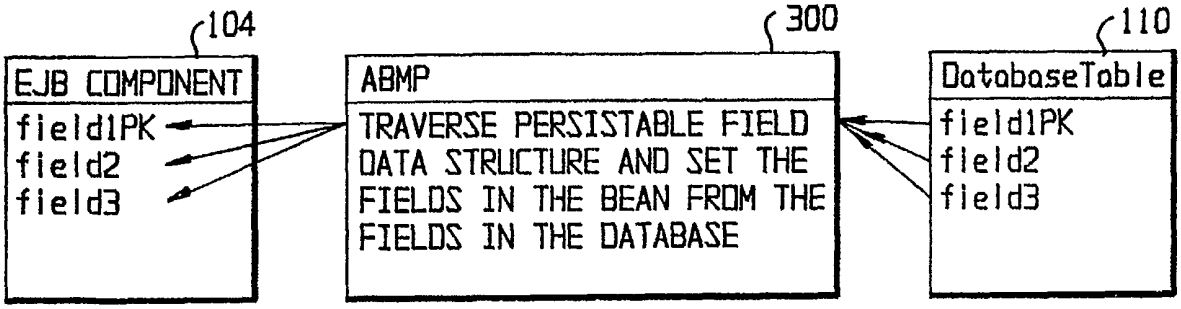


FIG. 3C

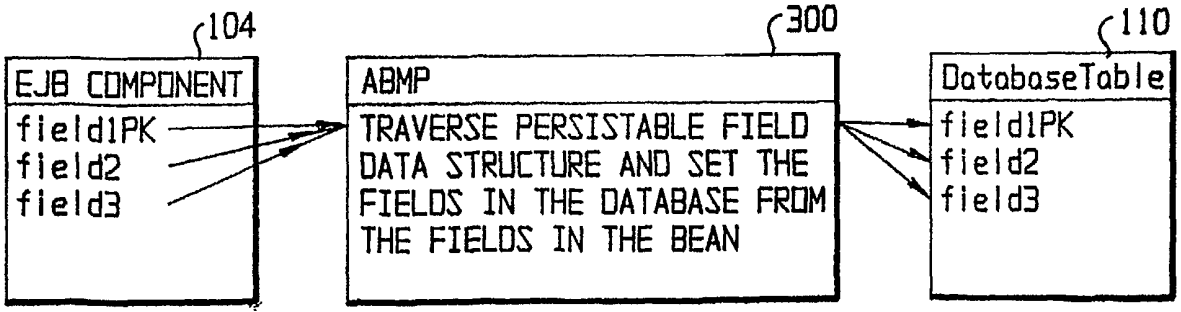


FIG. 3D

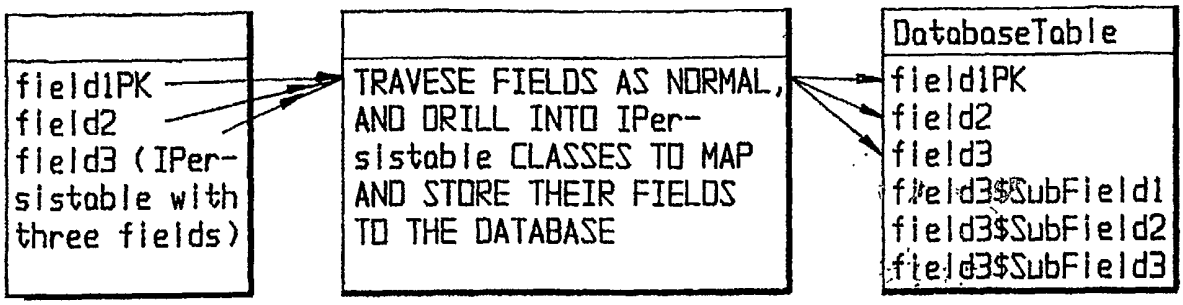

field1PK
field2
field3 (IPer-
sistable with
three fields)
TRAVESE FIELDS AS NORMAL,
AND DRILL INTO IPer-
sistable CLASSES TO MAP
AND STORE THEIR FIELDS
TO THE DATABASE
DatabaseTable
field1PK
field2
field3
field3$SubField1
field3$SubField2
field3$SubField3

FIG.4

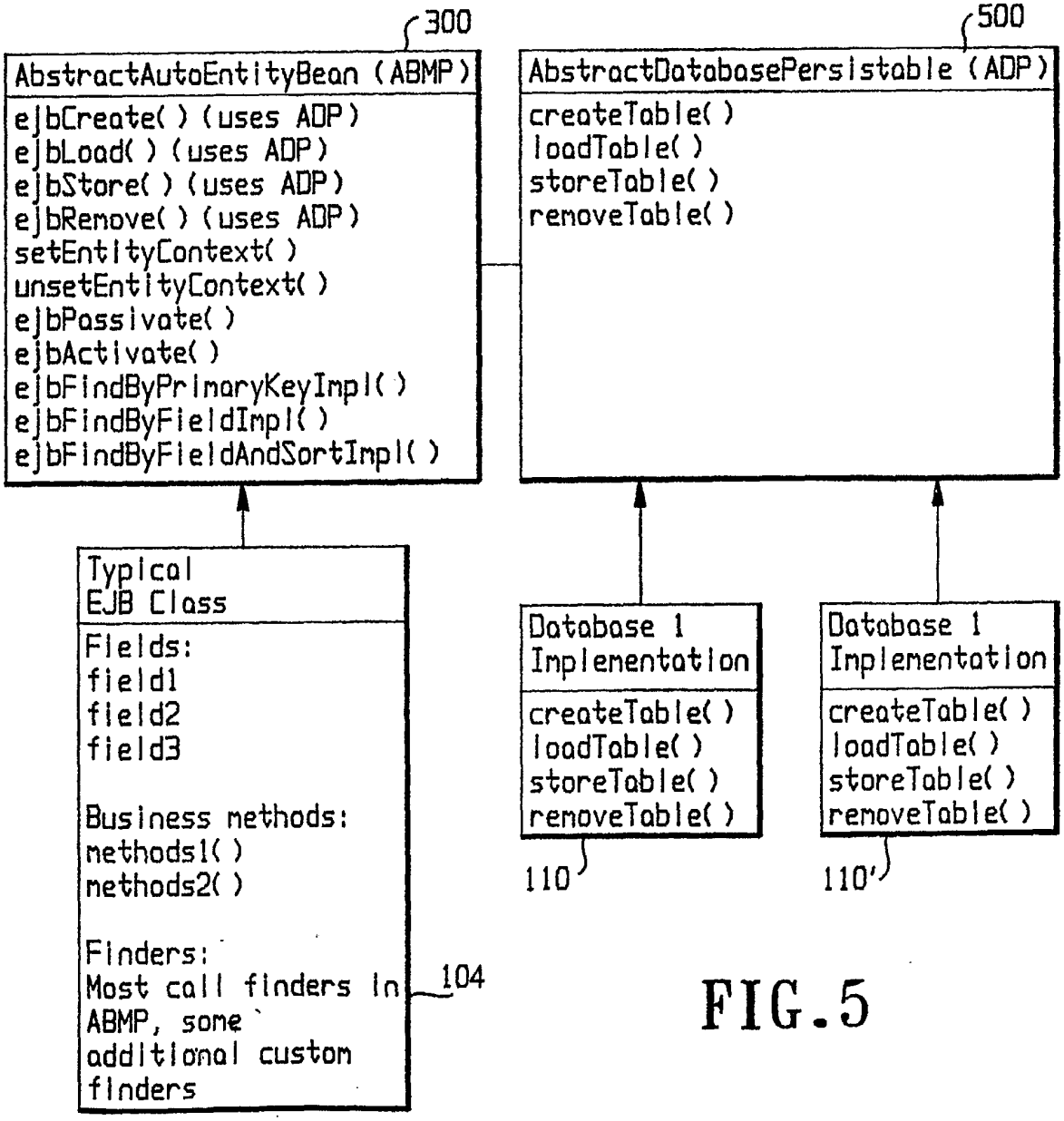

300
AbstractAutoEntityBean (ABMP)
ejbCreate( ) (uses ADP)
ejbLoad( ) (uses ADP)
ejbStore( ) (uses ADP)
ejbRemove( ) (uses ADP)
setEntityContext( )
unsetEntityContext( )
ejbPassivate( )
ejbActivate( )
ejbFindByPrimaryKeyImpl( )
ejbFindByFieldImpl( )
ejbFindByFieldAndSortImpl( )
500
AbstractDatabasePersistable (ADP)
createTable( )
loadTable( )
storeTable( )
removeTable( )
Typical
EJB Class
Fields:
field1
field2
field3
Business methods:
methods1( )
methods2( )
Finders:
Most call finders in
ABMP, some
additional custom
finders
104
Database 1
Implementation
createTable( )
loadTable( )
storeTable( )
removeTable( )
110
Database 1
Implementation
createTable( )
loadTable( )
storeTable( )
removeTable( )
110'

FIG.5