Tallinna Tehnikaülikool

Informaatikainstituut

Javal baseeruv objektide püsivuse kiht Jakamar

Bakalaureusetöö

Üliõpilane: Erki Suurjaak

Matrikli nr: 970772

Juhendaja: Tarmo Veskioja

Tallinn

2001

AUTORIDEKLARATSIOON

Deklareerin, et käesolev lõputöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud.

....

(kuupäev) (lõputöö kaitsja allkiri)

Java-Based Object Persistence Layer Jakamar

Abstract

My thesis introduces a concrete solution for achieving object persistence in a Java environment - the object persistence layer Jakamar.

Persistent objects are objects that exist beyond the lifetime of the application; this is typically achieved by storing them in some kind of a data store, most commonly a relational database. A basic task for software is producing, changing and viewing such data. There are several recurring problems that application programmers have to deal with when developing object-oriented applications that use relational databases as a persistence mechanism. The greatest problem is handling changes in the data structure - a simple alteration forces some reworking in both the application persistence logic and the database. Larger changes - e.g. switching the underlying database from one vendor to another - can require a great amount of work. Another concern is application design - the persistence logic of many applications is similar, differing mainly in the exact data structure. It would be favorable to exploit this similarity and produce an automated solution.

I set out to achieve the following goals:

- To propose a reusable generic approach to handling persistence logic - the *Persistence Broker* design pattern, which solves several problems when using relational database management systems in an application environment. This pattern represents a solution where business logic is separated from persistence logic to the degree where persistent objects are not aware that they are being stored and retrieved. The solution is easily customizable for the data structure of different applications.
- To present Jakamar, a component I had written in the Java programming language as an implementation of this pattern. Jakamar is an object persistence layer with the sole purpose of being a building block that is used for developing other applications. It provides fully automated persistence - the application programmer has no need for

writing any database access code, the component handles all data storage, retrieval and deletion by itself.

- To compare using Jakamar with using embedding database access logic directly in program code, and to compare Jakamar with other software products that follow the *Persistence Broker* pattern.

All the three goals were realized. I deemed the *Persistence Broker* to be a well-designed approach to persistence logic, and Jakamar to be a viable software component. Jakamar encapsulated the entire persistence logic of an application, and provided access to persistence via simple operations like *store(object)*, *delete(object)* and *retrieve(criteria)*. In addition to solving the problems listed above, using Jakamar yielded better modularized application structure, faster program development (as the programmer does not need to develop any database access code), and possibility for better performance on object retrieval.

Keywords: automated persistence, persistence layer, object-relational mapping, persistence broker.

Contents

1. Introduction

1.1. Background

A basic task for object-oriented software is producing, changing and viewing persistent objects. Persistent objects are objects that exists beyond the lifetime of the application; this is typically achieved by storing their data in some kind of a data store.

The most common persistence mechanism at the moment is a relational database management system [BibTec01]. Relational databases are an efficient and proven technology, they have widespread support in development languages and third party tools, and people are familiar with them. However, several problems arise in a combination of using an object-oriented language (like Java, SmallTalk or C++) as the development environment and a relational database as the persistence mechanism.

The greatest problem is getting locked into a proprietary technology. Although the language for accessing relational databases - Structured Query Language (SQL) - conforms to a standard, most database vendors add functionality to their product that extends the syntax of the language. This additional functionality is useful and rewarding, but produces a tight coupling between the database and the application. Should the underlying database change, then this change cascades into the application domain as well. For example, if the database management system is changed from Microsoft SQL Server to Oracle, then most probably the persistence logic in the application (the code that handles storing, deleting and retrieving persistent data) needs to be reworked to conform to the syntax and ideology of Oracle.

Another problem is the impact of changes in data structure. If the class structure changes in some way, then, besides the database, this triggers changes in the persistence logic of the application as well. For example, if a field type is changed from integer to floating-point value, then even this minor modification induces a change in the persistence logic.

Final issue is not exactly a problem, but a point of concern nonetheless - the persistence logic in applications could be automated. The principal difference between the persistence logic of two applications is the exact structure of data, but the basic

persistence operations - like retrieving values from the database and setting them to objects, and retrieving object field values and storing them in the database - are alike.

1.2. Objective

This thesis offers a solution to these concerns. First, I introduce the *Persistence Broker* design pattern[*] for a reusable persistence layer that separates application logic from persistence logic. The basic idea for such software was introduced in Scott W. Ambler's whitepaper "Mapping Objects to Relational Databases" [Ambl00a].

Secondly, I present a component that I wrote as an implementation of such software - the Jakamar object persistence layer. Jakamar, written in the Java programming language, provides application objects with transparent persistence. With transparent persistence, persistence of objects is provided automatically and the logic for performing persistence operations is expressed in the Java language, without the client programmer needing to know anything about databases and SQL. Jakamar encapsulates all the persistence logic that an application needs, and gives access to persistence via simple operations like *store(object)*, *delete(object)*, and *retrieve(criteria)*.

Thirdly, I compare the benefits of using the *Persistence Broker* pattern with the basic approach to persistence logic - embedding database access directly into the application code - and judge the usefulness of the pattern. I also compare Jakamar with similar publicly available Java software that at least in some terms follow the pattern and offer similar functionality, and I judge the viability of Jakamar.

[*] A design pattern is a description of associated objects and classes that are customized to solve a general, recurring design problem in a particular context [Gam95].

1.3. Thesis Composition

The thesis is divided into three main parts:

The *Persistence Broker* Pattern

Describes the problem domain and offers the solution. Using pattern notation has the advantage of laying down the idea in a uniform clear-cut way that many developers are familiar with. For information on patterns, see *1.4.1 Pattern Notation*.

The Jakamar Object Persistence Layer

Describes the Jakamar object persistence layer, a component that follows the *Persistence Broker* pattern. Gives an overview of its functionality, architecture, and other aspects like security and configurability.

Benefits and Tradeoffs

Describes benefits and tradeoffs of a persistence broker instead of the basic approach to persistence - embedding database access code directly in the application logic. Describes also other components and frameworks that at least in some aspects follow the *Persistence Broker* pattern. They are compared to Jakamar in terms of functionality and performance.

1.4. Notation

1.4.1. Pattern Notation

The notation to describe the problem and the solution is pattern notation.

A design pattern is a description of associated objects and classes that are customized to solve a general, recurring design problem in a particular context [Gam95]. Patterns provide a way to capture and reuse expertise.

The pattern notation used follows the AG Communication Systems Pattern Template [AGCS1].

The following parts of a pattern definition might need some explanation:

Problem	Gives a statement of the problem that this pattern resolves.
Context	Describes the context where the problem is found.
Forces	Describes the forces influencing the problem and solution. Forces basically elaborate the context. Forces can have positive and negative effect.
Solution	Gives a statement of the solution to the problem.
Resulting Context	Describes the context of the solution. Can include new problems that appear as a result of applying the pattern.
Rationale	Explains the rationale behind the solution.

1.4.2. Diagram Notation

The notation used to mark up static and dynamic views of the system architecture - class and interaction diagrams - is UML [UML97]. The definitions below have been taken from the above reference.

a) Classes

A **class** is a set of objects that share the same attributes, operations, methods, relationships, and semantics.

An **interface** specifies the externally visible behaviour of a class, or object, including the signatures of the operations.

An **abstract class** is a class that cannot be directly instantiated.

A **concrete class** is a class that can be directly instantiated.

Figure 1-1. *Notations of an interface, an abstract class and a concrete class.*

b) Class members

A **member** is a part of a type or class denoting either an attribute or an operation. Square brackets ([]) for attribute type or operation parameter type denote an array.

An **attribute** is a named property of a type.

An **operation** is a service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the actual parameters that are possible.

A **constructor** can be regarded as a special operation that creates and returns a new instance of the class. Constructors have the same name as the class.

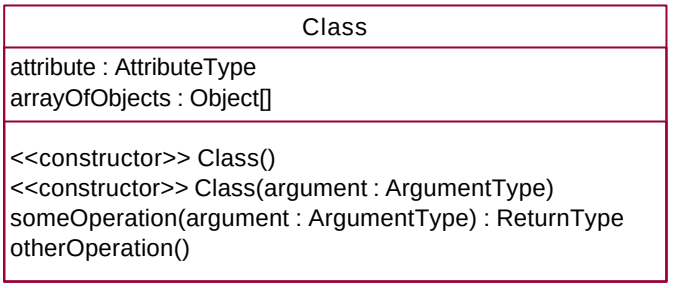

Figure 1-2. *Notation of class members.*

c) Class relationships

A **relationship** is a semantic connection among model elements.

Generalization relationships

Generalization is the taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed.

I**nheritance** is the mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior.

I**nterface inheritance** is the inheritance of the interface of a more specific element. Also known as the **implements** relationship.

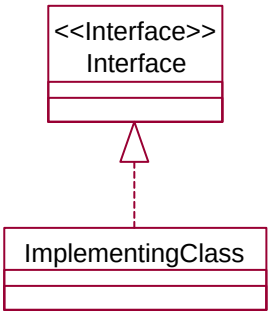

Figure 1-3. *Notation of the **implements** relationship.*

I**mplementation inheritance** is the inheritance of the implementation of a more specific element. Includes inheritance of the interface. Also known as the **extends** relationship.

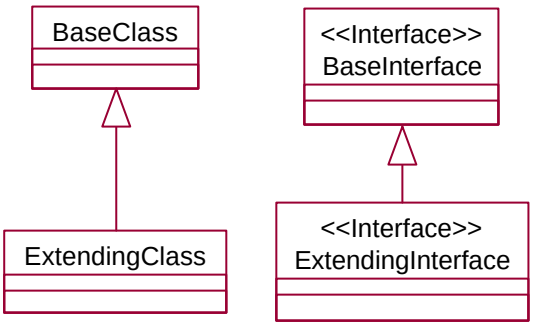

Figure 1-4. *Notation of the **extends** relationship.*

Association relationships

An **association** is a relationship that describes a set of semantic connections among a tuple of objects. Both sides of an association can have roles that identify the purpose or capacity wherein one class associates with another.

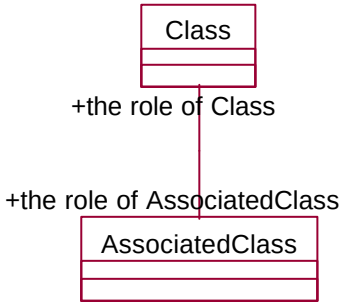

Figure 1-5. *Notation of the association.*

An **aggregation** is a special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part.

A **composition** is a form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes).

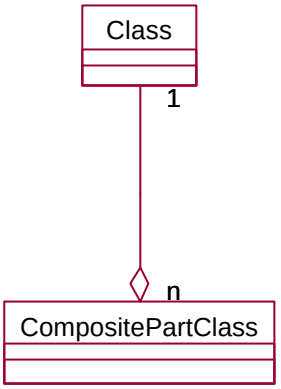

Figure 1-6. *Notation of t composition.*

d) Interaction diagrams

A **sequence diagram** is a graphical view of a scenario that shows object interaction in a time-based sequence - what happens first, what happens next.

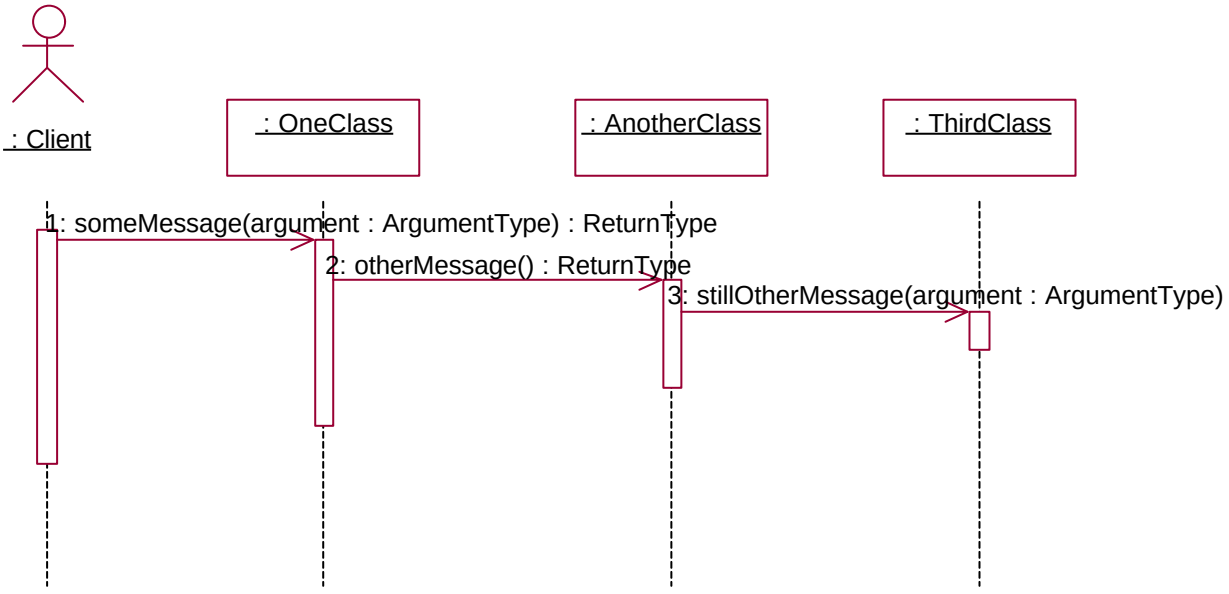

Figure 1-7. *Notation of a sequence diagram.*

A **collaboration diagram** is a diagram that shows object interactions organized around the objects and their links to each other. Unlike a sequence diagram a collaboration diagram shows the relationships among the objects. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

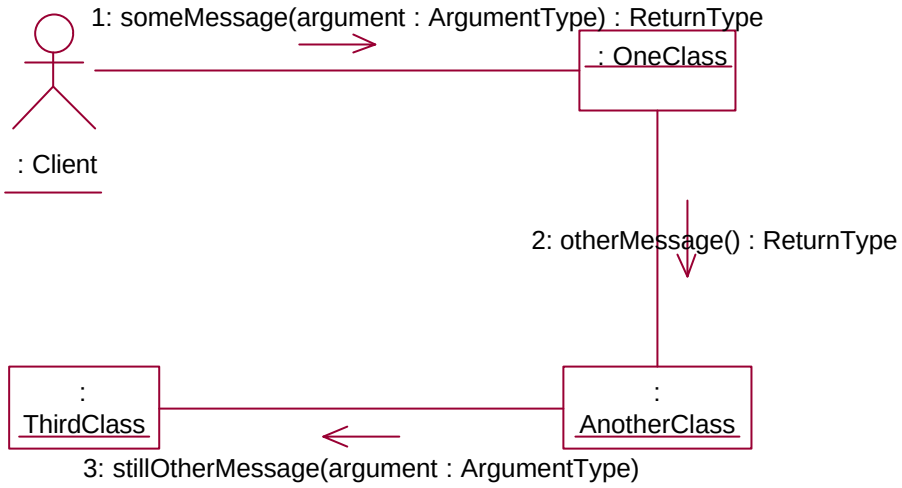

Figure 1-8. *Notation of a collaboration diagram.*

e) Activity diagrams

An **activity diagram** shows flow of control from activity to activity. An activity is an ongoing non-atomic execution that results in some action

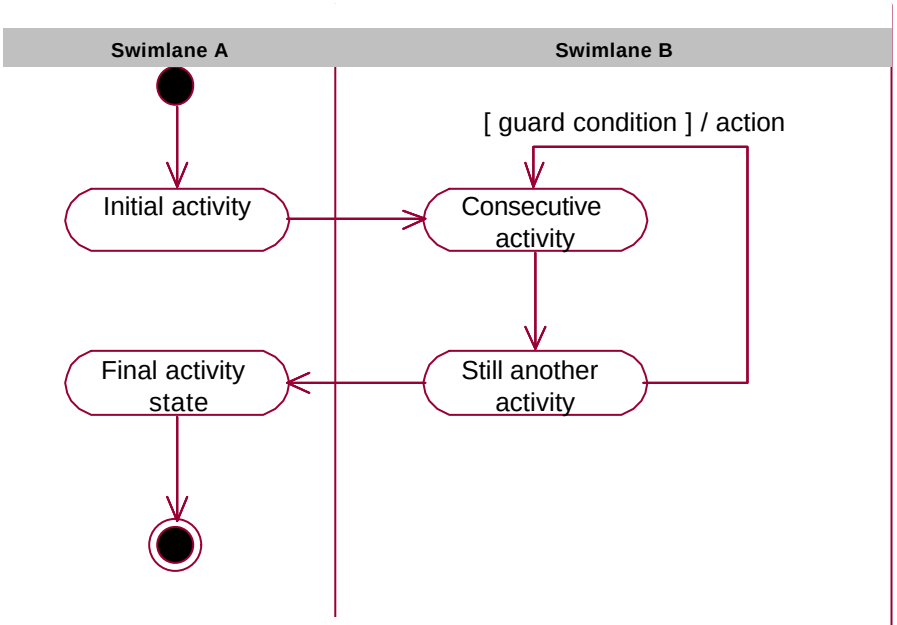

Figure 1-9. *Notation of an activity diagram.*

2. The *Persistence Broker* Pattern

Aliases: Persistence Manager.

2.1. Context

You are developing an application that needs to store the objects it works with. Most probably, the persistence mechanism used is a relational database.

Example: a web-based book database, displaying the entered authors and their books, and allowing the entry of new authors and books.

2.2. Problem

How to provide application objects with persistence, without hard-coding the system to use a proprietary technology that can be subject to change? What solution would be easily reusable in other applications?

2.3. Forces

- The most commonly used persistence mechanism at the moment is a relational database management system [BibTec01].
- Object databases are relatively new and uncommon, and do not have the support relational databases have [BibTec01].
- Every database vendor provides a slightly different syntax for functionality outside the standard SQL, which gives greater power to the programmer, but enforces the use of a proprietary technology.
- Tailoring an application to a proprietary technology hinders portability, risks enslavement to the vendor and can cause great difficulties if the technology proves insufficient for new requirements.
- Embedding statements for data storage (e.g. SQL queries) in persistent classes is fast both in performance and initial development, but hinders portability and forces reworking of the classes in case of changes in the data structure. For example,

changing the field type of a class triggers a change of both the database and the persistence logic. Or, a database table column's data type is changed from INTEGER to BIT, but the object's field type remains a boolean.

- Adding persistence into the classes themselves is not always possible - they might be third party components.
- Persistence methodology is often very similar across applications, differing only in the exact data structure, which makes it a candidate for automation and reuse.
- Developing a generic, reusable persistence mechanism takes a lot more time, effort and skills than embedding an application-specific mechanism in the application.

2.4. Solution

Create a component that is able to provide application objects with persistence. The component is not directly associated with the persistent classes and is therefore fully reusable. Configuring the component to handle new persistent classes is dynamical, done at run-time, from a configuration file. The component's functionality equals that of a relational database, providing a query mechanism, relational integrity, and transaction support, which are significant in many applications. The provided persistence is transparent - the client programmers do not need to be aware how class instances are made to persist, all the client programmer is aware of is that calling the methods of the component - e.g. *store*, *delete,* and *retrieve* - provides access to persistence. Extending the software to support other kinds of data stores besides relational databases should be possible with no impact on existing code. Persistent classes need not be persistence aware, i.e. they do not need to extend a certain class - accessing the object fields is done using reflection[*].

Note that in case of a programming language that does not support reflection, it is impossible to implement the solution in the current form. Languages that support reflection include Java, C#, and Visual Basic. Languages that do not yet support reflection include C, C++, and Delphi. In case of such languages, some workaround must be used -

[*] Reflection is the process of inspecting a class for meta-information - information about its fields and methods, calling methods, accessing fields dynamically, etc [McM97].

for example, developing a small adapter between the application and the component, which is familiar with the application classes and knows how to interact with the component.

Overview of the functionality of the *Persistence Broker*:

- stores objects in the data store
- deletes objects from the data store
- retrieves objects from the data store, via potentially complex queries
- supports referential integrity - when storing an object, its related objects will be stored as well; when deleting an object, its related objects will be deleted with it
- supports transactions
- supports inheritance*

2.5. Resulting Context

Application programmer is not concerned with the specific persistence mechanism used. He has been provided with an easily used, transparent persistence mechanism that can be used with different databases. The problem of achieving object persistence, which in some applications can comprise half of the total effort, has become a non-issue. The programmer can concentrate on other aspects of the application.

New problem: performance overhead. The persistence mechanism of the application has become notably more complicated, adding at least one, possibly several layers of logic between persistent classes and the data store. This incurs an additional hit on performance. Besides, the reflection mechanism of the programming language can be slow.

New problem: decrease in the functionality of the persistence mechanism. With relational databases, the programmer can execute extremely complex queries just as

* Inheritance is an example of the object-relational mismatch [Ambl00a]. Persistent classes can inherit from other persistent classes, extending them with new attributes. This inheritance needs to be supported in a relational database as well. This can be implemented either storing the entire inheritance tree in one table, storing each class in a separate table including all the attributes, or storing each class in a separate table including only the attributes specific to that concrete class. When retrieving an object of a persistent class that has persistent extensions, the extensions will have to be queried as well.

readily as simple storage and retrieval statements. With a persistence broker, other workarounds must be used.

2.6. Rationale

By encapsulating persistence logic in a separate place and providing transparent persistence, the application programmer need not concentrate on the specifics of one data store, which should not be in the scope of most applications, but can instead concentrate on the problems of the application domain.

Most of the databases available today offer additional functionality besides that provided by the ANSI SQL standard. They provide row identification mechanisms, special functions, triggers, and stored procedures, all of which are useful. But using them in an application produces a tight coupling between the application and the database. If the database is never changed during the lifetime of the application, then there is no concern. If however, it happens that there is need to make the application use a different database - e.g. the current one imposes some limits that have been finally reached, or the software licence has expired and acquiring another licence from the same vendor is unacceptable - then the application programmer is faced with a possibly huge task of refactoring the application persistence mechanism. Utilizing a reusable component capable of handling any kind of a database instead of making use of proprietary technology, this domain of problems has become a total non-issue.

Now that the persistence broker is ready, it can and will be used in many applications. The application programmer can utilize it without paying attention or even being aware of whether the underlying data store is a database from the same vendor, from another vendor, or an altogether different mechanism, like an XML file. If a minor change is made in the underlying data store, the programmer need not change any part of the application If a bigger change is made, then there is no persistence logic code to change, only the persistence broker configuration file.

2.7. Known Uses

The general idea for a *Persistence Broker*-like architecture was introduced by Scott W. Ambler [Ambl00b]. However, his work did not propose the broker in pattern form, and specified the classes to be persistence aware.

ObJectRelationalBridge, an open source Object/Relational mapping tool, utilizes this pattern (http://objectbridge.sourceforge.net/).

2.8. Example

2.8.1. Example Architecture

Two example architectures are given, one for an application that uses direct coupling with the database, another for an application that uses a persistence broker.

Figure 2-1 shows the sample architecture for an application that has directly embedded database access logic.

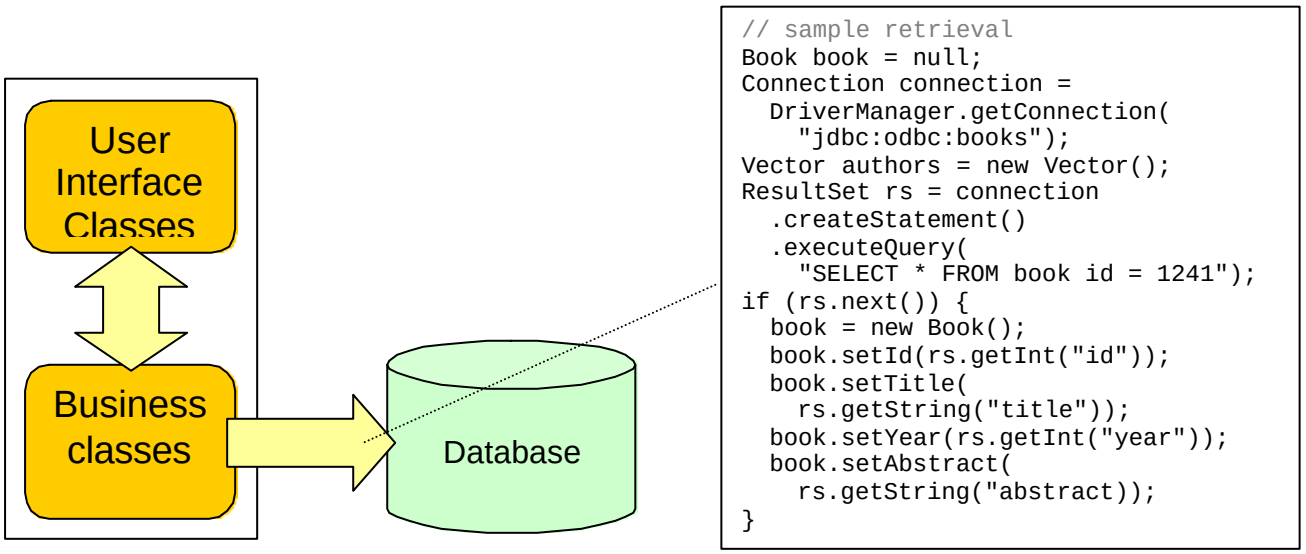

Figure 2-1. *Sample architecture of an application having directly embedded database access logic.*

Figure 2-2 shows the sample architecture for an application that uses a persistence broker.

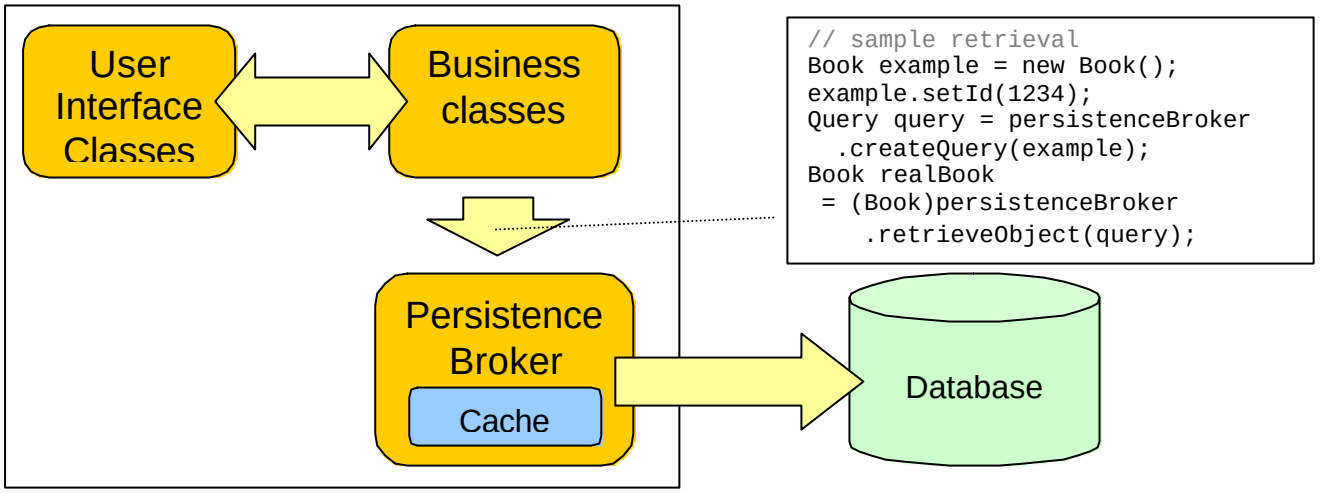

Figure 2-2. *Sample architecture of an application having directly embedded database access logic.*

2.8.2. Example Code

Two examples are given, one for an application that uses direct coupling with the database, another for an application that uses a persistence broker.

Figure 2-3 contains the business classes of the application, Book and Author.

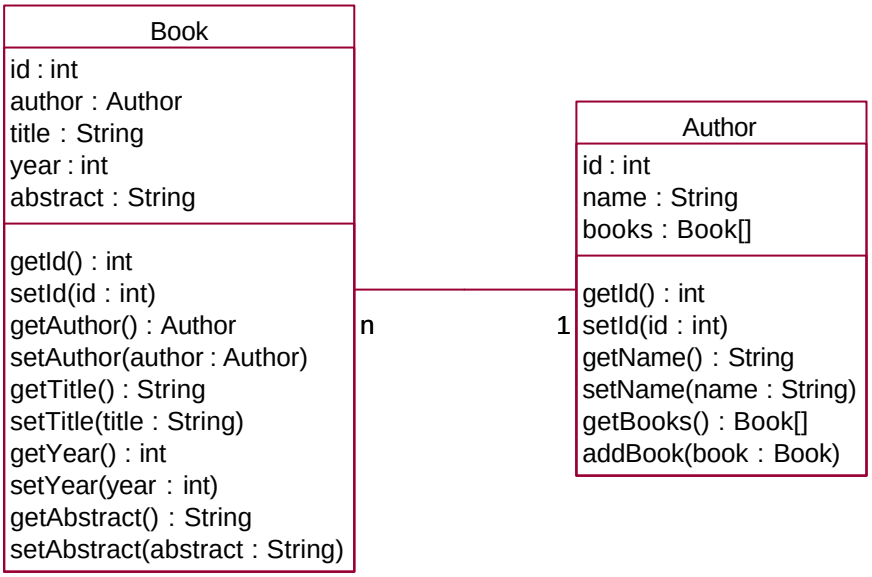

Figure 2-3. *Sample application classes that need persistence.*

Figure 2-4 contains the database table diagram for the application.

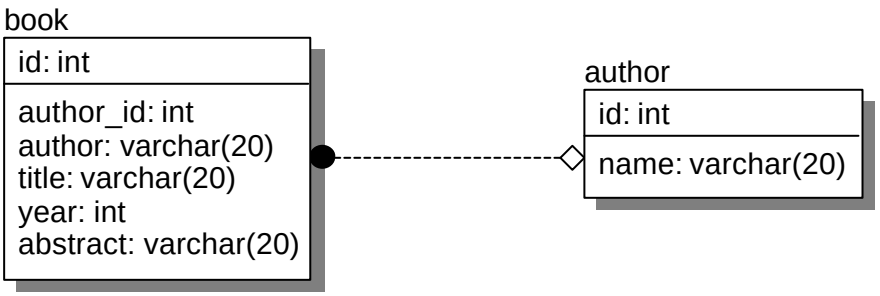

Figure 2-4. *Database table diagram for a sample application.*

2.8.2.1. Direct Database Access

Sample code in Java using direct database access.

```
// An example that retrieves a set of Author instances from the database
// and prints them out.
public class DatabaseTest {
  public static void main(String[] args) throws Exception {
    // Create a connection to the database
    Connection connection =
      DriverManager.getConnection("jdbc:odbc:books");
    Vector authors = new Vector();
    // Query the table author, where the name contains 'donald'
    ResultSet rs = connection.createStatement().executeQuery(
      "SELECT * FROM author WHERE name = '%donald%'");
    while (rs.next()) {
      Author author = new Author();
      author.setId(rs.getInt("id"));
      author.setName(rs.getName("name"));
      // Query the table book for all the books of this author
      ResultSet subRs = statement.executeQuery(
        "SELECT * FROM book WHERE author_id = " & author.getId());
      while (subRs.next()) {
        Book book = new Book();
        book.setAuthor(author);
        book.setId(subRs.getInt("id"));
        book.setTitle(subRs.getString("title"));
        book.setYear(subRs.getInt("year"));
        book.setAbstract(subRs.getString("abstract));
        author.addBook(book);
      }
      authors.add(author);
    }
    connection.close();

    // Iterate over the retrieved authors and print them out
    for (int i = 0; i < authors.size(); i++) {
      Author author = (Author)authors.get(i);
      System.out.println("Author: " + author.getName() + ". Books: ");
      Book[] books = author.getBooks();
      // Iterate over the books of the author and print them out
      for (int j = 0; j < books.length; i++) {
        System.out.println("  " + books[j].getName() + ". Published in "
                           + books[j].getYear());
      }
    }
  }
}
```

2.8.2.2. Using a Persistence Broker

Figure 2-5 contains the Persistence broker classes:

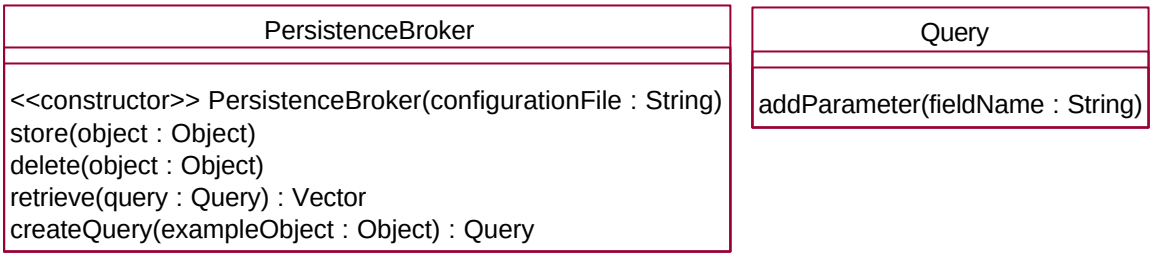

Figure 2-5. *Sample classes for an interface to Persistence Broker.*

Sample code in Java using a persistence broker.

```
// An example that retrieves a set of Author instances from the
// persistence broker and prints them out.
public class BrokerTest {
  public static void main(String[] args) throws Exception {
    // Construct the persistence broker from a configuration file
    PersistenceBroker broker = new PersistenceBroker("conf.xml");
    // Build the query and execute it
    Author example = new Author();
    example.setAuthor("%donald%");
    Query query = broker.createQuery(example);
    query.addParameter("author");
    Vector authors = broker.retrieve(query);

    // Iterate over the retrieved authors and print them out
    for (int i = 0; i < authors.size(); i++) {
      Author author = (Author)authors.get(i);
      System.out.println("Author: " + author.getName() + ". Books: ");
      Book[] books = author.getBooks();
      // Iterate over the books of the author and print them out
      for (int j = 0; j < books.length; i++) {
        System.out.println("  " + books[j].getName() + ". Published in "
                           + books[j].getYear());
      }
    }
  }
}
```

As visible from the example above, using a Persistence Broker yields cleaner application code, and forces no change in the persistence logic should the data structure change, while using direct coupling with the database will bring about a change for every small change.

3. The Jakamar* Object Persistence Layer

3.1. Introduction

Jakamar is an object persistence layer that implements the *Persistence Broker* pattern. It is a component that provides access the persistence via the simple *PersistenceBroker* interface. Jakamar is not an independent application - its sole purpose is to be a reusable software package that is utilized by other applications, being a building block that is used for developing stand-alone applications.

Jakamar is implemented in Java. Forces for choosing Java as the implementation language include:

- Java has extensive API for database access (called JDBC).
- Java is a common platform for server-side applications, which rely heavily on databases and persistent data.
- Java is platform independent.
- Java supports reflection.
- The author is intimately familiar with Java.

3.2. Functionality

Jakamar provides all of the functionality required of a persistence broker (see *2.4 Solution*), except support for transactions and inheritance. These two aspects are intended to be added during further development (see *3.11 Further Development*).

Feature overview:

- uses relational databases for data storage, automating object-relational mapping

* It is a common practise to give software products names that often are unrelated to the product domain. For example, Ant is a popular Java compiling tool, and Tomcat is a Java web component container. I encountered the name "jakamar" in Jules Verne's *Mysterious Island* - it referred to a very rare and beautiful bird having a sharp long beak. It was edible, but no delicacy. I believe the Latin name for it is *Galbula viridis Lath*, but I have found no authorative reference for this.

- information about mapping Java classes onto underlying database tables is read from a configuration file. The content of the configuration file is provided by the client programmer.
- stores objects in the data store
- deletes objects from the data store
- retrieves objects from the data store, via potentially complex retrieval criteria
- can retrieve multiple objects with one operation
- supports one-to-many relationships between classes - when retrieving an object, its related objects can be retrieved automatically
- supports referential integrity - when storing an object, its related objects will be stored as well; when deleting an object, its related objects will be deleted with it
- caches persistent objects, which can yield a dramatic increase in retrieval speed
- can access multiple databases
- persistent classes do not have to persistence aware, e.g. they do not have to inherit from a specific class
- supports composite identities
- can perform any kind of conversion between object fields and table columns in both types and values - for example, storing a string in the database, but a boolean in the object (e.g. "male" and "female" in the database and isMale in the object)
- custom SQL can be specified

The main interface of the component is the *jakamar.PersistenceBroker* interface. Client programmers can obtain an instance of this interface by letting it to be created from a configuration file that holds the information for mapping Java classes onto a data store. For further information on configurability, see *3.10 Configuration*.

Example places where Jakamar can be used:

- web applications for information systems, which basically are database frontends and therefore rely heavily on stored data
- desktop applications, that need to store user preferences
- applets for informations systems, acting as database frontends

- any kind of application where there is need to persist objects in databases

Figure 3-1 presents the architecture of a sample application using Jakamar.

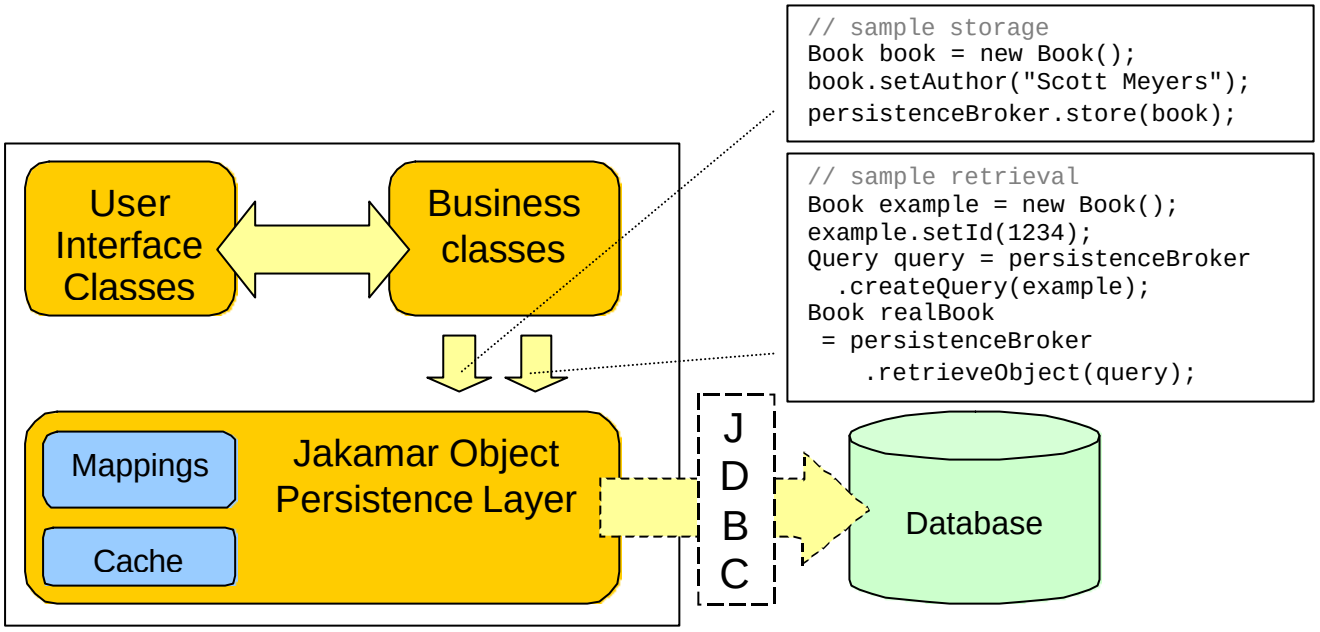

Figure 3-1. *Architecture of a sample application using Jakamar.*

Jakamar has been designed with regard to future support for other data stores besides relational databases. For example, if another package is provided that implements the PersistenceBroker interface and uses XML files for data storage, then it can be integrated seamlessly with the existing code. Nothing changes for the client programmer - persistence brokers of different type will be obtained in exactly the same way.

3.2.1. Broker Instantiation

The main interface that client programmers work with is *PersistenceBroker*. Persistence brokers are created via a factory class and built from configuration files. If an instance built from the specified configuration file is already available, the client programmer is handed the existing instance.

An example of obtaining a PersistenceBroker instance:

```
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
```

This approach has the benefit of supporting the seamless integration of new types of persistence brokers discussed above, as the configuration file specifies what kind of a persistence broker to use.

3.2.2. Storing an Object

Storing an object is very straightforward. The client programmer only needs to send a message to the persistence broker and the object gets stored. If the object's identity is unspecified - which is the case if the object has been transient so far - it is assigned a unique identity. For more on object identities, see *3.3 Object Identities*.

An example of storing an object:

```
// Obtain a Persistencebroker instance
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
// Create a new object and set some of its attributes
Book book = new Book();
book.setTitle("Effective C++");
book.setAuthor("Scott Meyers");
// Store the object
broker.store(book);
```

3.2.3. Deleting an Object

To be able to delete an object from the underlying data store, the only information needed to know is its identity. For more on object identities, see *3.3 Object Identities*.

An example of deleting an object:

```
// Obtain a Persistencebroker instance
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
// Let the user input the identity of the Book to delete
System.out.print("Enter the id of the Book to delete: ");
BufferedReader in
  = new BufferedReader(new InputStreamReader(System.in));
String input = in.readLine();
int id = Integer.parseInt(input);
// Create an example object and set its identity to that of
// the object to delete
Book book = new Book();
book.setId(id);
// Delete the object
broker.delete(book);
```

3.2.4. Retrieving Objects

Jakamar introduces the *Query* object - an object which represents query criteria and is used for object retrieval and criteria. The client programmer obtains an instance from the persistence broker, is able to perform some additional operations on it (e.g. setting parameters and ordering information), and gives the query to the persistence broker to be executed.

There are three types of queries (in order of ascending complexity):

Query by identity

The criteria for selection is object identity. This query is used if the object identity is known. A query by identity utilizes the object cache and can provide a remarkable performance gain if the object has already been materialized - performing object retrieval in practically zero time. This query retrieves at most one object. For more information on object identities, see *3.3 Object Identities*.

Example of use:

```
// Obtain a Persistencebroker instance
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
// Create the object with the identity
Book example = new Book();
example.setId(2412);
// Create the Query instance
QueryByIdentity query = broker.createQuery(example);
// Retrieve the result of the query
Object result = broker.retrieveObject(query);
```

Query by example

An example object is provided, with the fields that hold example values being specified as parameter fields. Every object of this class that shares the same field values is selected. As this query can retrieve multiple results, the results can be ordered on specified attributes.

Example of use:

```
// Obtain a Persistencebroker instance
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
// Create the example object
Book example = new Book();
example.setAuthor("Bjarne Stroustrup");
String[] params = {"author"};
// Create the Query instance
QueryByExample query = broker.createQuery(example, params, null);
// Order the results ascendingly by title and then descendingly by year
query.orderBy("title", false);
query.orderBy("year", false);
// Retrieve the results of the query
Collection results = broker.retrieveCollection(query);
```

Query by criteria

Allows the specification of complex and structured query criteria. Criteria can include "less than", "not equal to", "greater or equal" etc. As this query can retrieve multiple results, the results can be ordered on specified attributes.

Example of use:

```
// Obtain a Persistencebroker instance
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
// Construct the criteria
Criteria criteria = new Criteria();
// The last argument specifies whether the criterion is to be added
// to the rest of the criteria using AND or OR as the operator
criteria.add("author", "James Gosling", Criteria.EQUAL, false);
criteria.add("title", "%Java%", Criteria.LIKE, true);
Criteria subCriteria = new Criteria();
subCriteria.add("author", "Bjarne Stroustroup", Criteria.EQUAL, false);
subCriteria.add("title", "%C++%", Criteria.LIKE, true);
criteria.add(subCriteria, false);
// Order the results by author
OrderBy[] orderBys = {new OrderBy("author")};
// Create the Query instance. The query selects all Books that
// either have Bjarne Stroustroup as the author and "C++" as a substring
// in the title, or James Gosling as the author and "Java" as a
// substring in the title.
QueryByCriteria query = broker.createQuery("Book", criteria, orderBys);
// Retrieve the results of the query
Collection results = broker.retrieveCollection(query);
```

3.3. Object Identities

An object identity (OID) is an identifier that, assigned to a persistent object, uniquely identifies the object. In a relational database, tables have key columns. The values of the key columns of a row make up the identity of a row. If objects, mapped to relational databases, are to have identities, then these key columns (which usually have no business meaning) must intrude into the object classes (where they normally are not present). Classes are assigned attributes that map onto key columns in the database table.

The intrusion is necessary for creating relationships between objects. If at run-time a persistent object has a reference to another persistent object, then when the objects are made persistent, this reference needs to be stored, and the only way to do it is to via OIDs. For example, a master class Person has a detail class Address, and a Person object can contain several Address objects. When storing the Person object and its Address objects, the OID of the Person object is stored as a foreign key in the database table the Address class maps onto. When the Person object is materialized again, its Address objects are also materialized by using the Person's OID.

Another use for OIDs in Jakamar is in caching - to cache an object, the object is mapped to its OID, and cache lookup is performed using the OID. If the objects of a class have no identity, then they cannot be cached and the performance gain from caching is lost.

The OID class for object-relational persistence in Jakamar is the *jakamar.jdbc.JdbcIdentity* class.

3.3.1. Identity Generation

New object identities are created when an object, that so far has been transient, is stored. If an object has no identity set yet, a new unique identity is generated and assigned to it.

Currently there are three identity generation strategies available in Jakamar. All these generation strategies assume the key columns to be numerical. Arbitrary implementations, however, are free to use columns of any type. For information on configuring the

persistence layer to use an arbitrary identity generator, see *3.10.5 Generating Object Identities*.

Identity Generation from SELECT MAX

New valid values for key columns are obtained by performing a SELECT MAX function on an integer column and incrementing the result. The advantage of this approach is simplicity. The disadvantage is inefficiency - this demands a database access call for every identity generated. Also, obtained key values are unique over the table only, not the entire database.

The class *jakamar.jdbc.IdentityGeneratorFromSelectMaxBuilder* builds generators of this type.

Identity Generation from Counter

There is a table in the database that holds new valid values for key columns. The table can have an additional column that specifies the name of the table the values are for. The advantage of this approach is that key values can be unique over the entire database. The disadvantage is inefficiency - this demands two database access calls for every identity generated - one for selecting values, other for setting new values (which are obtained by incrementing the selected values) that later calls can use. Also, the table can become a performance bottleneck, as it is accessed for every identity generation.

The class *jakamar.jdbc.IdentityGeneratorFromCounterBuilder* builds generators of this type.

Identity Generation from High/Low [Ambl00a]

There is a table in the database that holds a range of new valid values for key columns. There are two values, HIGH and LOW, with HIGH specifying the starting value of the range and LOW specifying the length of the range. For example, with HIGH being 10000 and LOW being 50, the values generated will fall into the range 10000 - 10049. If the range is exhausted, two calls are made to the database - one for obtaining a new range (which, if other applications have not updated the range, is 10050-10099), the other for

updating the range in the database for later calls (setting HIGH to 10100). The table can have an additional column that specifies the name of the table the values are for.

The advantages of this approach are that key values can be unique over the entire database, and that it is efficient - a database access call is needed only if the range has been exhausted. The disadvantage of this approach is that as the range might not be exhausted during the lifetime of the identity generator object, it leaves gaps in the sequence of used values, thereby forcing possible values to run out faster. Where this becomes a concern, setting a smaller value for LOW, with the tradeoff of poorer performance, can mitigate it.

The class *jakamar.jdbc.IdentityGeneratorFromHighLowBuilder* builds generators of this type.

3.4. Object-Relational Mapping

The mapping of classes onto an underlying database is done using specific mapping classes. There are three items that need mapping:

- mapping the persistent classes themselves
- mapping class attributes
- mapping association relationships between classes

Mapping classes

Classes map onto tables, with one class per table. From Jakamar's point of view, there is no difference between ordinary tables and view tables.

The class that embodies class mapping is *jakamar.jdbc.JdbcClassMapping*.

Mapping attributes

Class attributes map onto table columns, with one attribute per column. The class that embodies class mapping is *jakamar.jdbc.JdbcFieldMapping*.

Mapping relationships

Jakamar supports unidirectional relationships - relationships which can be traversed in only one direction. In databases, relationships are implemented via primary and foreign keys - one part in the relationship holds the primary key value of the related part in a foreign key column. In the object realm, relationships are implemented as object references - one object has a field which points to the related object (or an array field, pointing to multiple related objects). When a persistence operation is performed with the object the relationship starts in, the operation can also cascade over the relationship, forcing the persistence layer to process the related objects as well. For example, if class Person has a one-to-many relationship to class Address (Address objects belonging to a Person), then deleting a Person object will force the deletion of the related Address objects as well, thus maintaining relational integrity in the database. All persistence operations - storing, deleting, retrieving - can be cascaded.

The class that embodies relationship mapping for object-relational persistence is *jakamar.jdbc.JdbcRelationshipMapping*.

3.5. Architecture

3.5.1. Conceptual View

Conceptual view gives an overview of the conceptual architecture of the system - what basic parts it has and how do they interact.

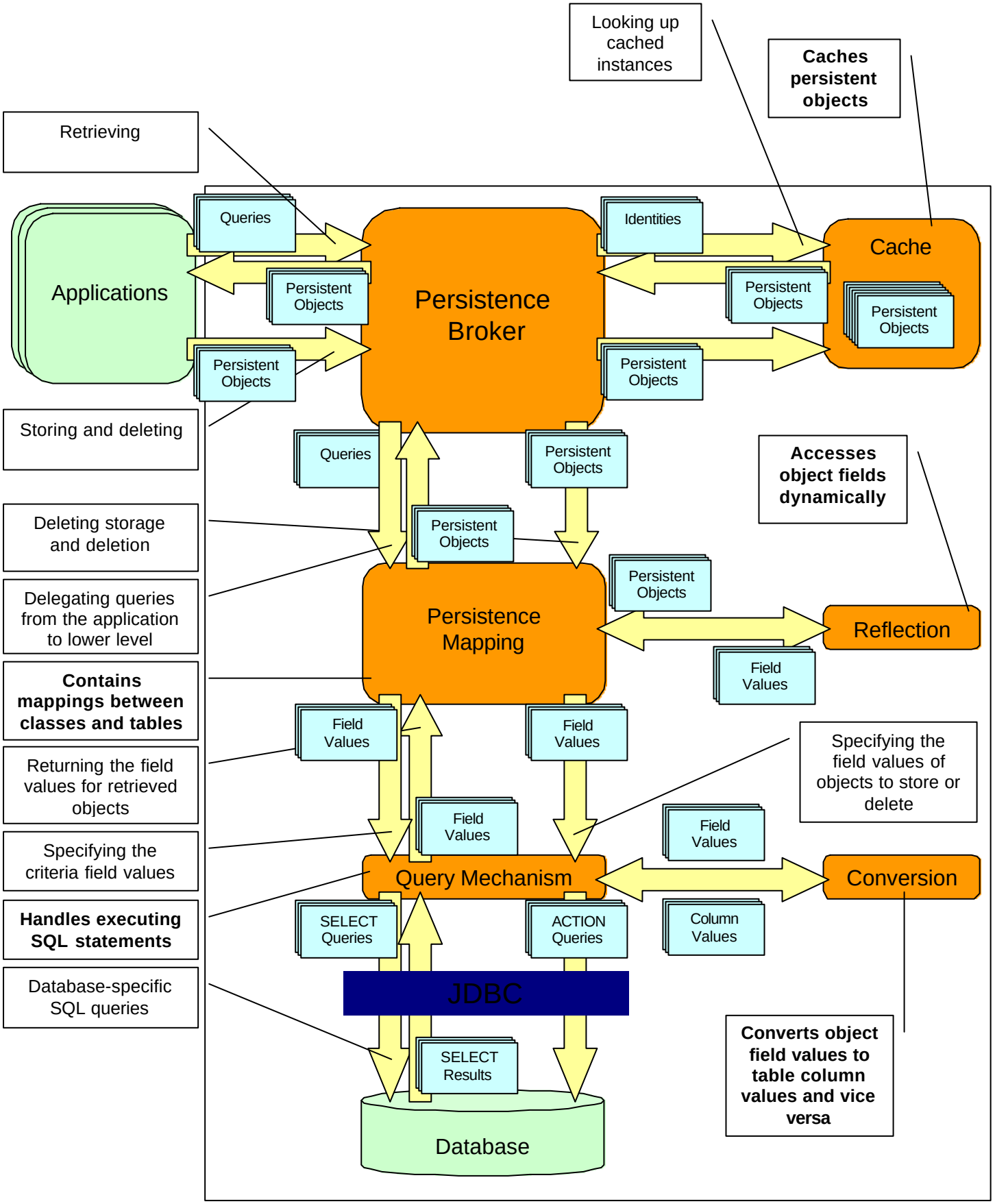

Figure 3-2. *Conceptual structure of Jakamar.*

3.5.2. Static View

The static view models relationships between classes - inheritance and associations.

The persistence layer is divided into packages. Packages group classes by commonality - for example package jakamar.mapping contains classes for mapping Java classes onto a data store.

3.5.2.1. Package jakamar

Contains the classes that the user of Jakamar will probably work with. In this way, Jakamar can be used with a single package import statement in code (`import jakakar.*;`).

Criteria

Is used for creating detailed and structured query criteria, used in conjunction with QueryByCriteria. For an example of use, see *3.2.4 Retrieving Objects.*

Criterion

Represents a single criterion, part of a Criteria. This is used internally by Criteria to represent all the selection conditions added to the object.

ConfigurationException

An exception thrown by *PersistenceBrokerFactory* indicating that there was an error when constructing a persistence broker from the configuration file - either the file could not be accessed or it contained malformed syntax.

DataStoreException

An exception thrown by the persistence layer indicating that an error occurred while accessing the data store. For example, the connection to the database was severed.

Identity

Represents an object identity. For further information on identities, see *3.3 Object Identities*.

InvalidQueryException

An exception indicating that a query contains invalid syntax or values. For example, if the client programmer has specified criteria for a field that does not exist or has no persistence mapping.

NestedPersistenceException

An exception containing a nested exception and indicating that an error has occurred in the persistence layer. This exception is used to allow the client programmer access to the original error encountered.

OrderBy

Represents ordering information for one query field. This is used internally by ordered queries (like QueryByExample and QueryByCriteria).

PersistenceBroker

The main interface of the persistence layer that client programmers work with. Provides functionality for storing, retrieving and deleting persistent objects.

PersistenceBrokerBuilder

Builds PersistenceBroker objects from the data fed to it by an instance of `jakamar.helpers.XmlHandler`. This is used internally by the persistence layer.

PersistenceBrokerFactory

Creates and vends PersistenceBroker objects. This is the class client programmers use to obtain an instance of PersistenceBroker.

PersistenceException

An exception indicating that an error has occurred in the persistence layer. This is the superclass for all persistence exceptions thrown by Jakamar.

Query

The basic Query interface for object retrieval. A generic query has basically no behaviour, but is used as a concept.

QueryByCriteria

A query that has detailed criteria. For an example of use, see *3.2.4 Retrieving Objects.*

QueryByExample

A query that selects objects by example values. When using this query, an object is provided, with fields set to desired values. For an example of use, see *3.2.4 Retrieving Objects.*

QueryByIdentity

A query that selects an object by its identity. This can be used if the identity of an object is known. This query is probably faster than other queries, as it utilized the object cache. For an example of use, see *3.2.4 Retrieving Objects.*

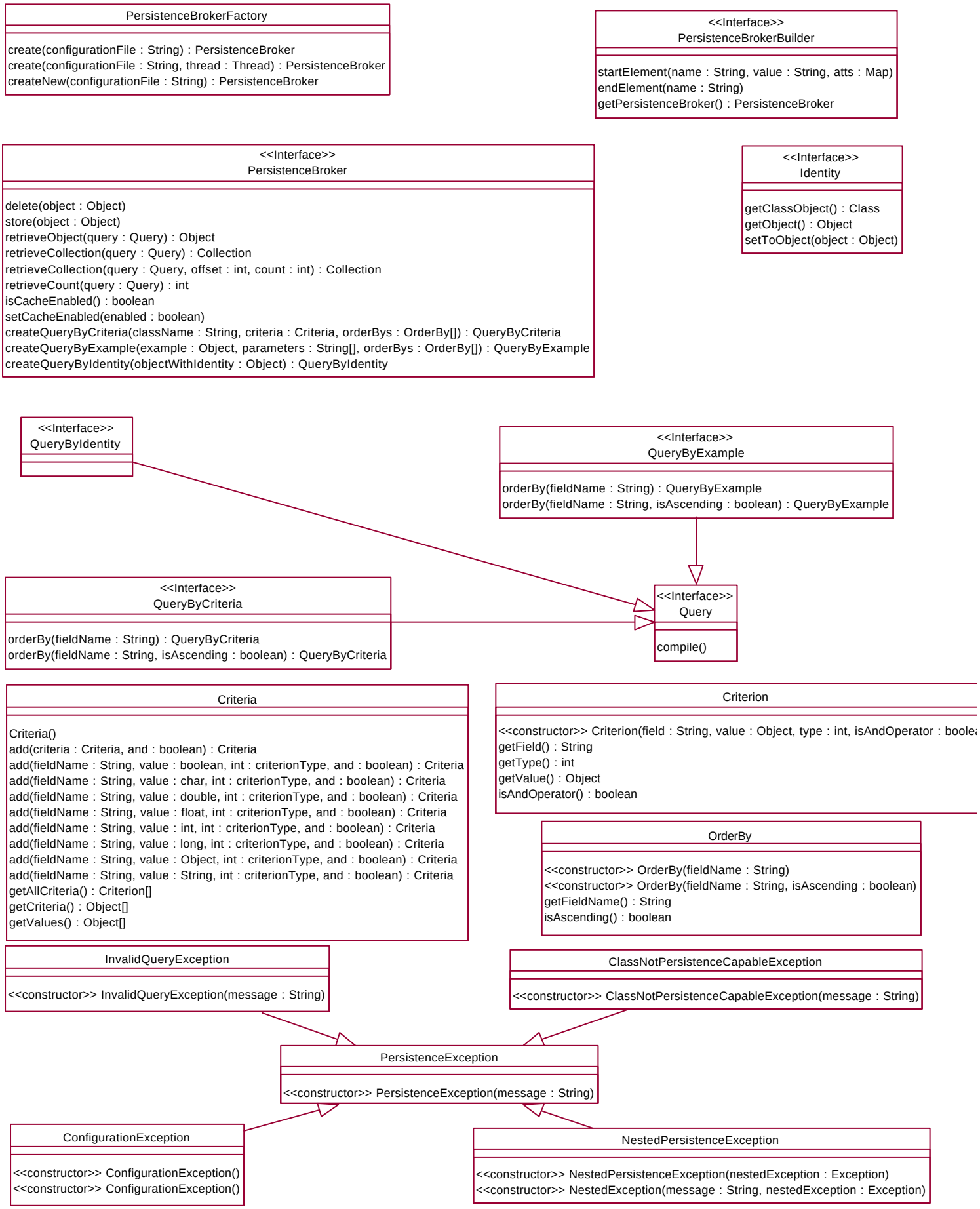

Figure 3-3. *Package jakamar - classes and inheritance.*

3.5.2.2. Package jakamar.mapping

Contains interfaces for persistence mapping. These interfaces specify the behaviour that persistence mappings for any kind of data store - not just the current implementation on relational databases - will have to implement.

ClassMapping

Contains information for mapping a Java class onto a persistent data store.

FieldMapping

Contains information for mapping a Java class field onto a persistent data store.

OneToManyRelationshipMapping

Contains information for maintaining a unidirectional one-to-many relationship between two classes.

OneToOneRelationshipMapping

Contains information for maintaining a unidirectional one-to-one relationship between two classes.

RelationshipMapping

Contains information for maintaining a unidirectional relationship between two classes, with a cardinality of 1 on the master side.

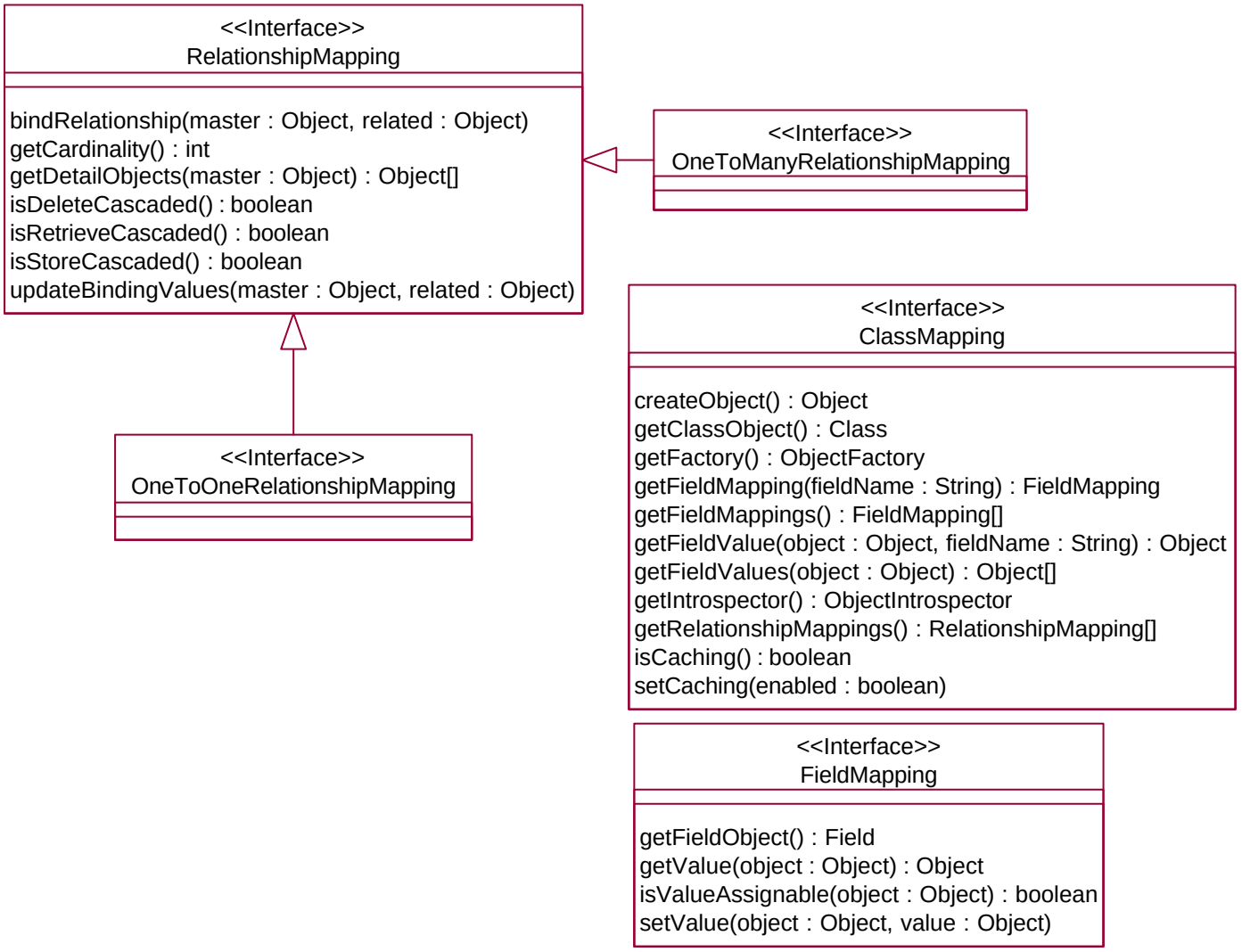

Figure 3-4. *Package jakamar.mapping - classes and inheritance.*

3.5.2.3. Package jakamar.helpers

Contains classes that are used internally by the persistence layer.

Cache

Caches persistent objects. Caching can yield notable results in object retrieval speed. This is provided as a pluggable interface - client programmers can configure the persistence framework to use an arbitrary implementation, if the default implementation provided by Jakamar is unsatisfactory. For further information on configuring, see *3.10 Configuration*.

ClassMappingBroker

Vends ClassMapping objects. This is just a container for ClassMapping objects to simplify the process of obtaining a class mapping for a specific class.

ErrorHandlingPolicy

Specifies an error handling policy to be used by the persistence layer. Top-level classes in the persistence layer use an error handling policy that dictates the action taken in case of an error. For further information, see 3.6 Error Handling.

FailFastErrorHandlingPolicy

An error handling policy with the ideology to fail quickly and cleanly.

ObjectFactory

Creates new objects. As persistent classes have to be instantiated dynamically, a stand-alone class for instantiation is useful. This is provided as a pluggable interface - client programmers can configure the persistence framework to use an arbitrary implementation, if the default implementation provided by Jakamar is unsatisfactory. For further information on configuring, see *3.10 Configuration*.

ObjectIntrospector

Provides access to object field values (inspecting and mutating the values). This is provided as a pluggable interface - client programmers can configure the persistence framework to use an arbitrary implementation, if the default implementation provided by Jakamar is unsatisfactory. For further information on configuring, see *3.10 Configuration*.

RobustErrorHandlingPolicy

An error handling policy with the ideology to never fail.

XmlHandler

Configuration file parser. This file is used internally by the persistence layer (by jakamar.PersistenceBrokerFactory) to parse configuration files and create persistence broker.

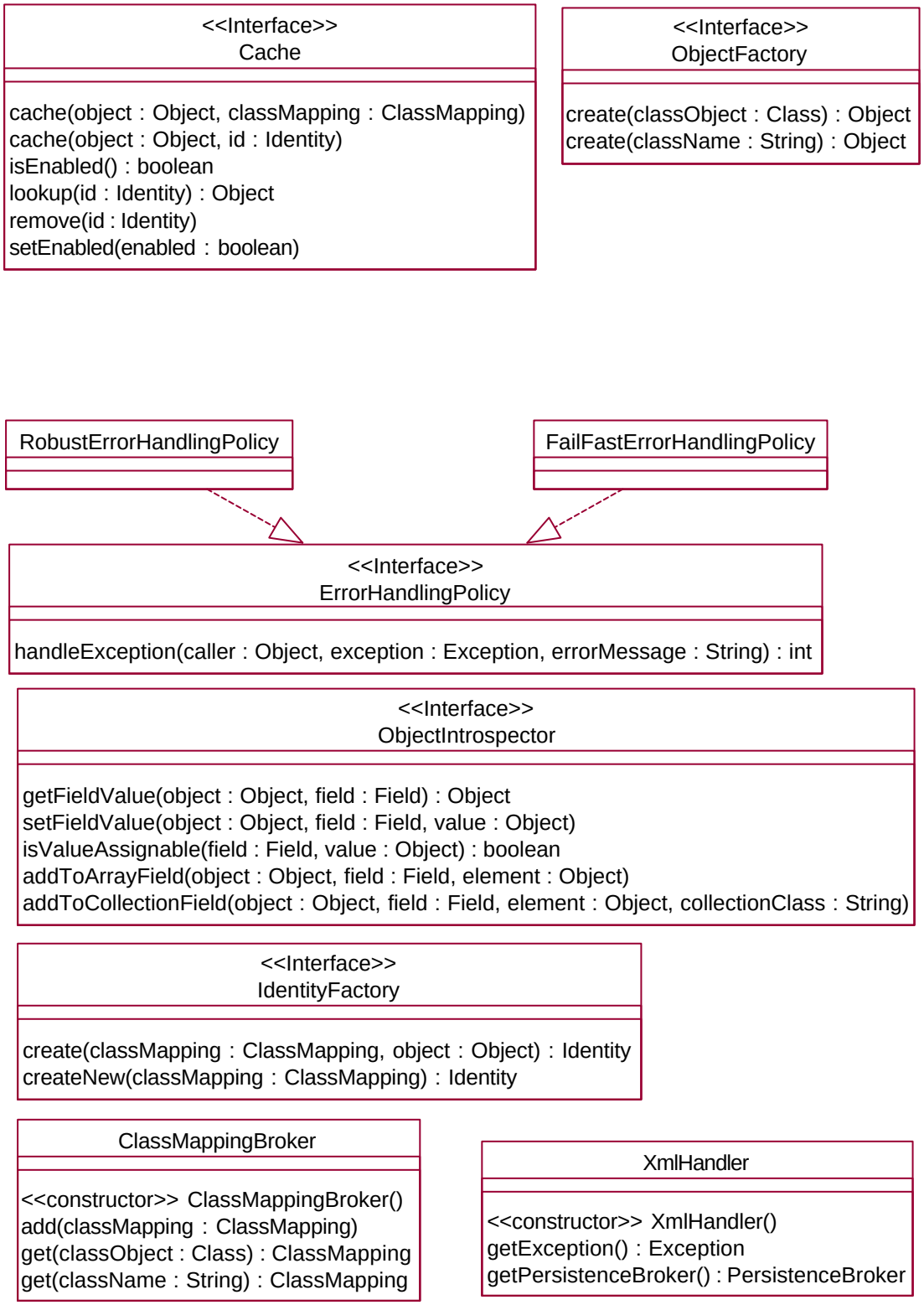

Figure 3-4. *Package jakamar.helpers - classes and inheritance.*

3.5.2.4. Package jakamar.jdbc

Contains an implementation of the persistence broker using relational databases as the data store.

As the classes in this package are quite numerous, they are divided into subdiagrams by commonality. If a class inherits from a class outside the jakamar.jdbc package, the package and the class have been imported onto the diagram.

Figure 3-5 contains the query classes for package jakamar.jdbc.

JdbcQuery

The JDBC query interface. This specifies the behaviour that all query classes of this package must follow, like returning an SQL WHERE statement.

JdbcQueryByCriteria

The JDBC implementation of jakamar.QueryByCriteria.

JdbcQueryByExample

The JDBC implementation of jakamar.QueryByExample.

JdbcQueryByIdentity

The JDBC implementation of jakamar.QueryByIdentity.

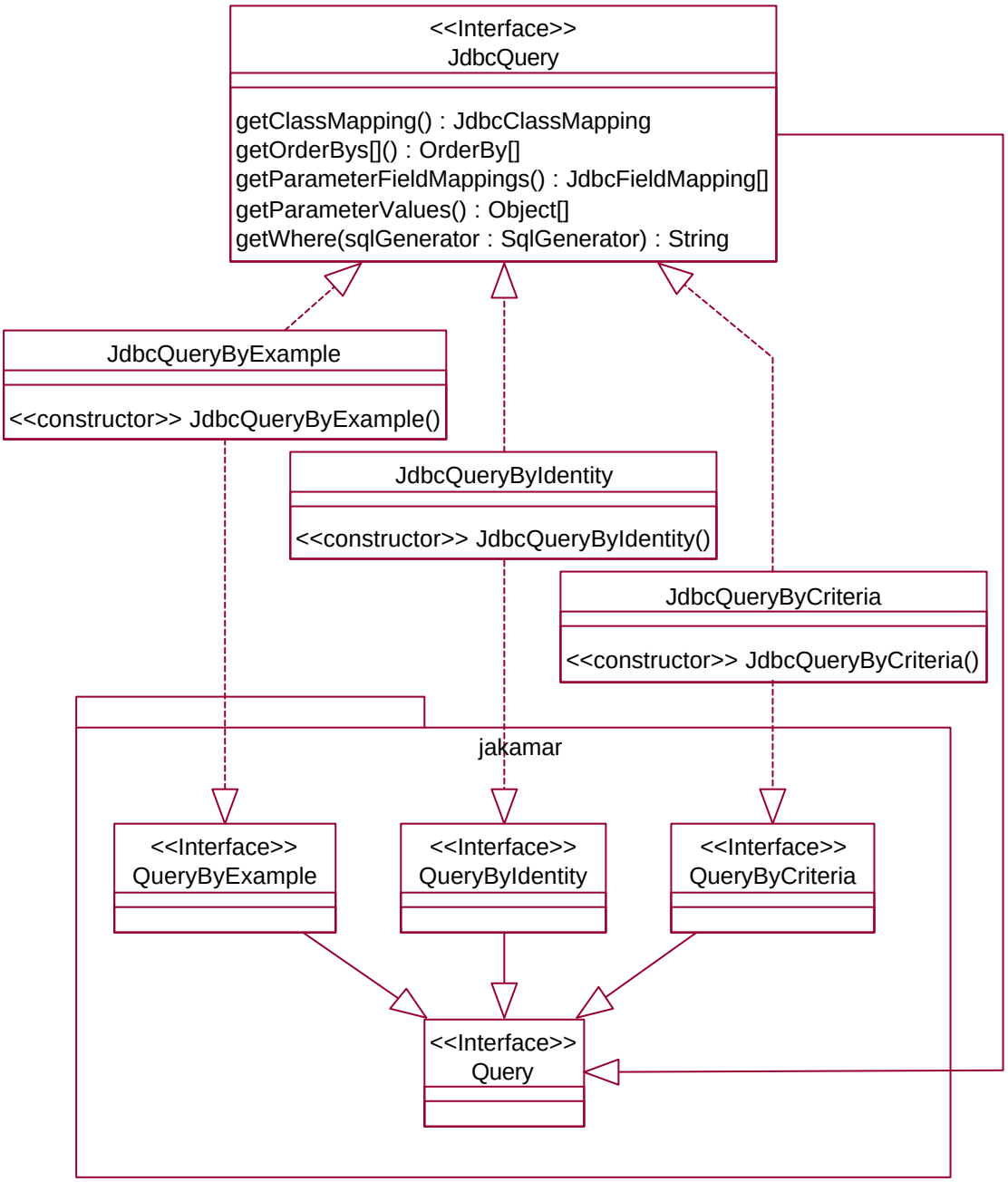

Figure 3-5. *Package jakamar.jdbc - query classes and inheritance.*

Figure 3-6 contains the classes that provide the mapping between Java classes and database tables. The key class is JdbcClassMapping - it contains JdbcFieldMapping and JdbcRelationshipMapping objects, has a reference to the Table its class maps onto, and to the Database the Table lies in.

Column

Holds information about a database table column. This information is used when creating SQL statements, and setting and getting column values from the database.

Database

Holds information about a database, and contains the Table objects for the tables of this database.

JdbcClassMapping

Contains information for mapping a Java class onto an underlying database table. Every class mapping refers to a Table object.

JdbcFieldMapping

Contains information for mapping a Java class field onto an underlying database table column. Every field mapping refers to a Column object.

JdbcOneToManyRelationshipMapping

The JDBC implementation of jakamar.mapping.OneToManyRelationshipMapping.

JdbcOneToOneRelationshipMapping

The JDBC implementation of jakamar.mapping.OneToOneRelationshipMapping.

JdbcRelationshipMapping

The JDBC interface of jakamar.mapping.RelationshipMapping. Specifies the behaviour that the relationship mappings of this package must implement, like creating a query that retrieves the related objects of a master object.

Table

Holds information about a database table, and contains the Column objects for the columns of this table.

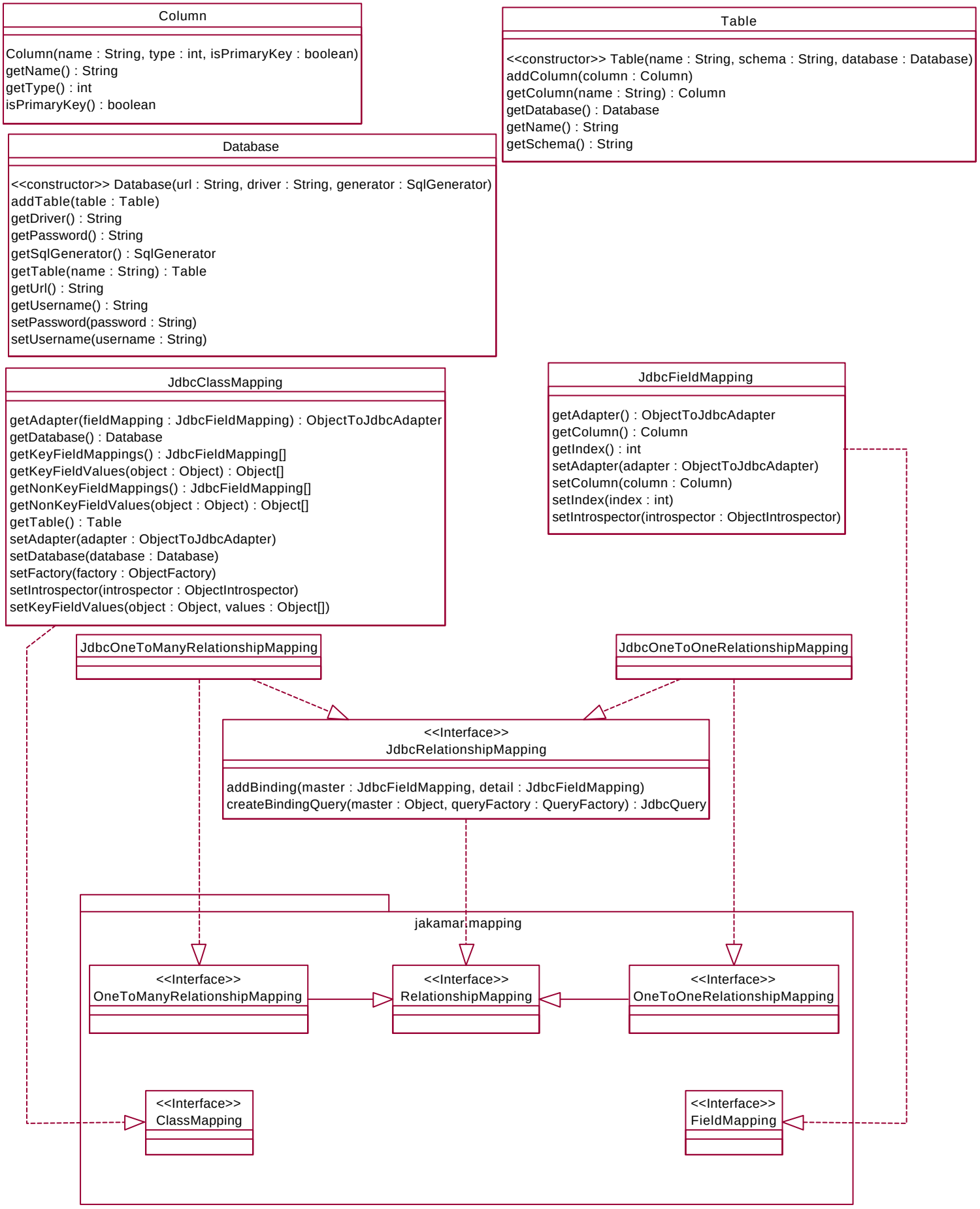

Figure 3-6. *Package jakamar.jdbc - persistence mapping classes and inheritance.*

Figure 3-7 contains classes that are close to persistence broker.

DbAccess

Handles all database access and persistence operations. The JdbcPersistenceBroker class is basically a front-end for this that wraps the PersistenceBroker interface around the class and provides support for caching and relationships.

IdentityGenerator

Generates new unique object identities. Instances of this interface are used by JdbcIdentityFactory to create new unique object identities. For further information on object identities, see *3.3 Object Identities*.

IdentityGeneratorBuilder

The interface that specifies the behaviour of all IdentityGenerator builders. This class and its implementations are used only when building a persistence broker from a configuration file. For further information on object identities, see *3.3.1 Identity Generation*.

IdentityGeneratorFromCounterBuilder

Builds identity generators that generate new unique identities by looking up the counter values from a special key-values table and incrementing the counter.

IdentityGeneratorFromHighLowBuilder

Builds identity generators that generate new unique identities by looking up the range of possible key values from a special key-values table and using items from the range as new identity values.

IdentityGeneratorFromSelectMaxBuilder

Builds identity generators that generate new unique identities by selecting the maximum values of the primary key columns and incrementing them by one.

JdbcIdentity

Represents the identity of a object mapped onto a database table.

JdbcIdentityFactory

Creates identities for persistent classes.

JdbcPersistenceBroker

The JDBC implementation of jakamar.PersistenceBroker. This class acts as a wrapper around the DbAccess class, providing additional functionality of caching and class relationships.

JdbcPersistenceBrokerBuilder

Builds JdbcPersistenceBroker objects. When building a persistence broker of this package, an instance of this class is created by jakamar.helpers.XmlHandler and data from the configuration file is delivered forward to the builder that knows what to do with it.

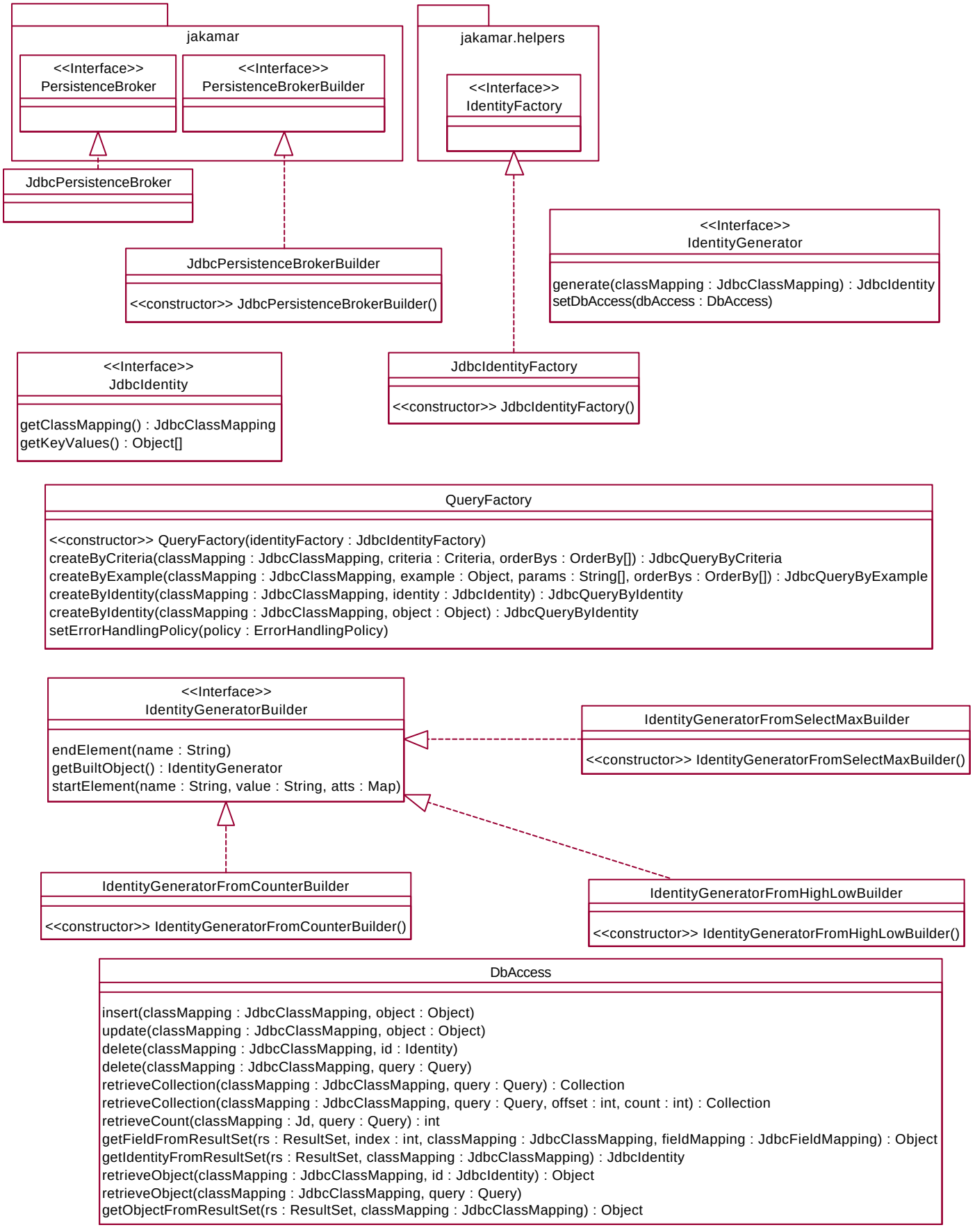

Figure 3-7. *Package jakamar.jdbc - classes and inheritance close to JdbcPersistenceBroker.*

Figure 3-8 contains the rest of the classes in the jakamar.jdbc package. These classes mostly deal directly with SQL and JBDC.

ClassStatementFactory

Constructs executable SQL statements for an individual class - statements returned by this class operate on one table and provide objects belonging to a certain class.

ConnectionManager

Manages database connections - hands out a connection to a specified database and expects the connection to be returned to be able to reuse it. Uses a pool of connections.

ObjectToJdbcAdapter

Provides persistent objects access to the JDBC storage and retrieval interfaces. Acts as an adapter between Java class fields and database table columns - it can be used for obtaining a value from the database that is fit to be set to an object field, and is able to insert the value of an object field into an SQL statement. This is a pluggable interface - client programmers can configure the persistence layer to use an arbitraty implementation, if the default implementation provided by Jakamar is unsatisfactory. For further information, see *3.10.1 Column-to-Field Conversions*.

SqlGenerator

Generates SQL syntax. All the SQL created for persistence operations comes from an SQL generator. Is Database-specific - every Database has an SQL generator instance. This is a pluggable interface - client programmers can configure the persistence layer to use an arbitrary implementation, if the default implementation provided by Jakamar is unsatisfactory. For further information, *see 3.10.4 Special SQL Syntax*.

StatementFactory

Factory class for creating executable Statement objects. It delegates all creation messages on to the suitable ClassStatementFactory object. This class is used by DbAccess

and acts as a convenient container for the possibly numerous ClassStatementFactory objects.

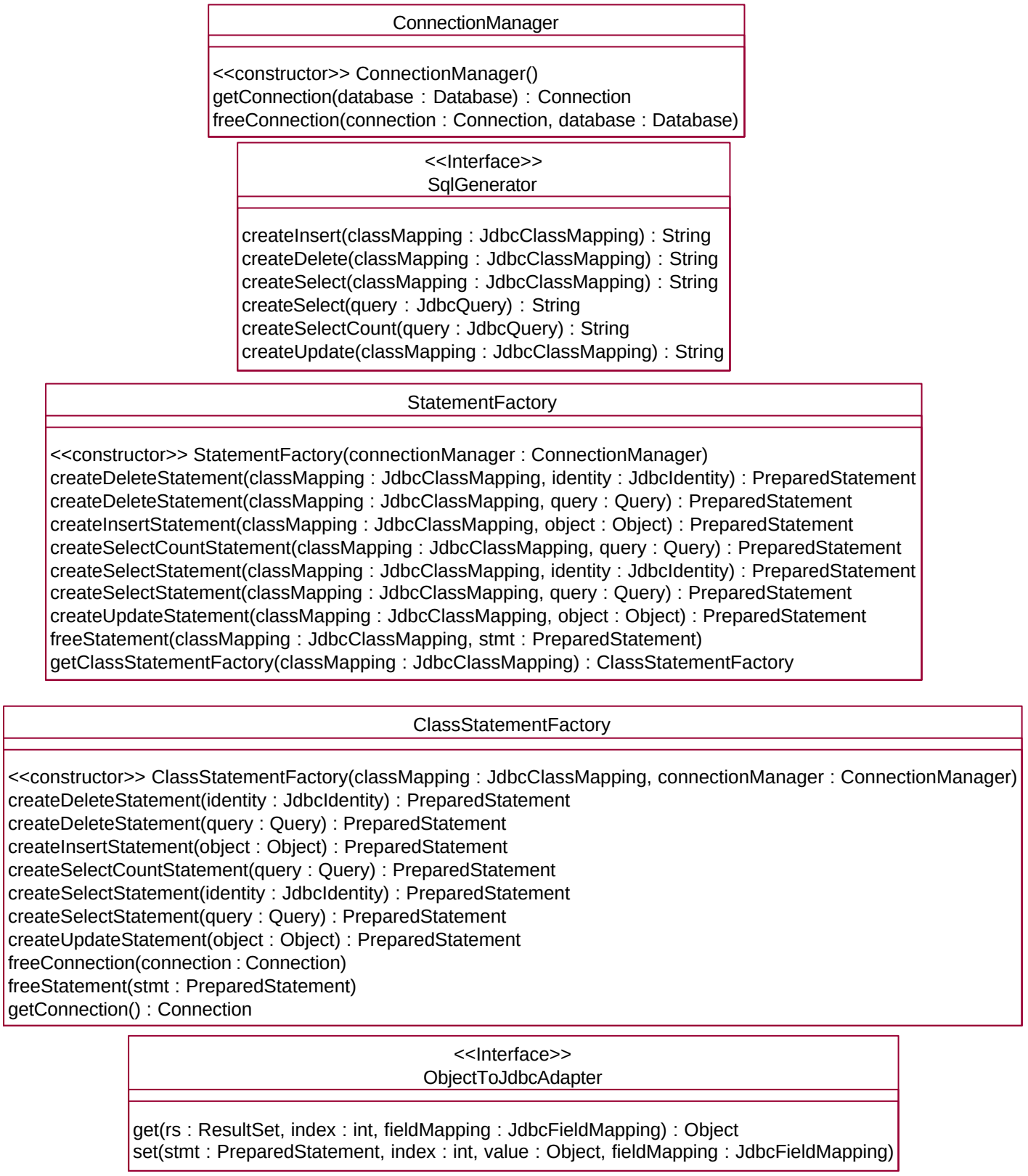

Figure 3-8. *Package jakamar.jdbc - classes and inheritance close to the JDBC framework.*

3.5.2.5. Associations

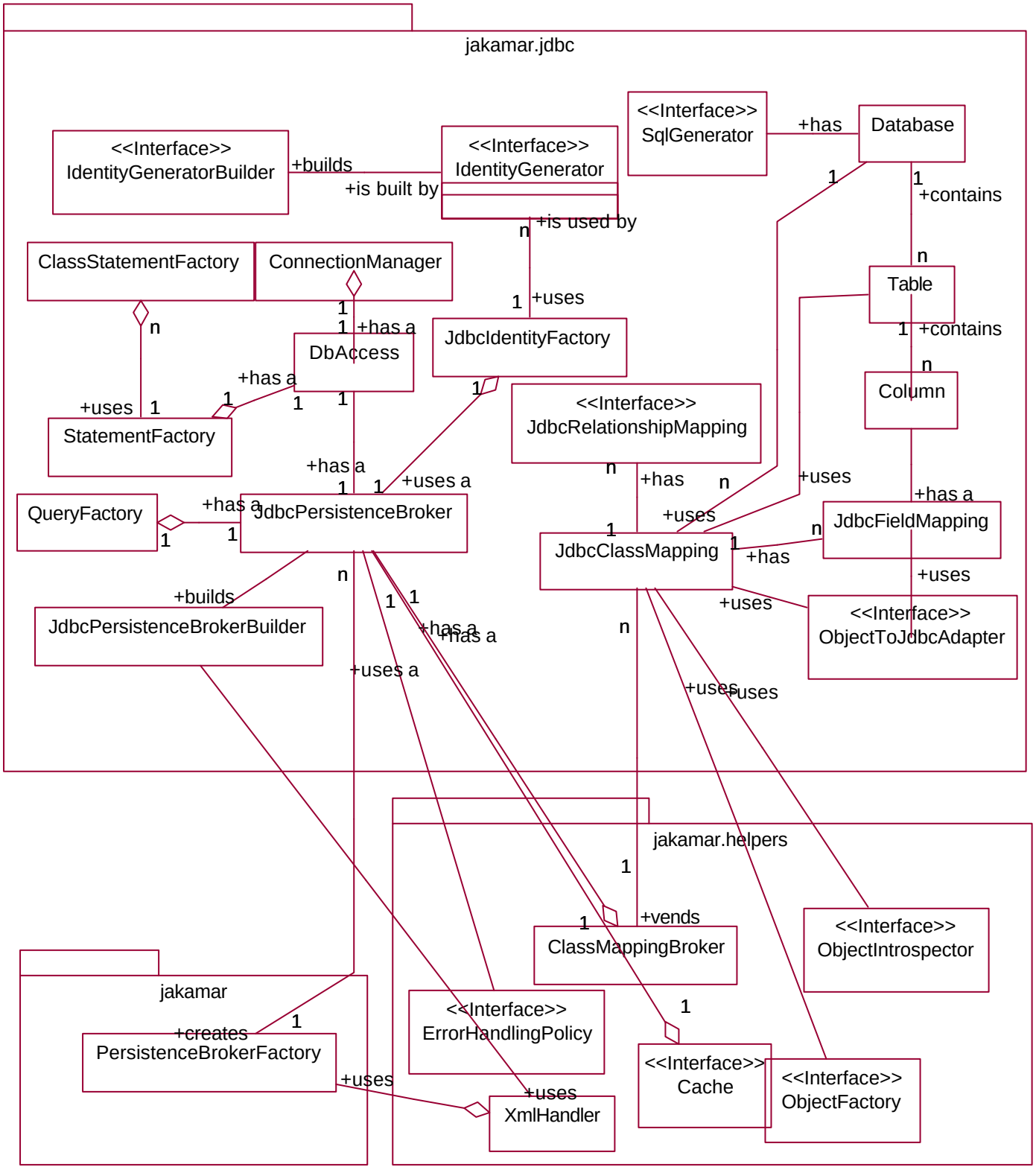

Figure 3-9. *Association relationships between classes across the jakamar packages.*

3.5.3. Dynamic View

The dynamic view models the interaction between class instances, marked up with sequence and collaboration diagrams. The main system operations - storing, deleting and retrieving - are modelled.

For every operation, there is a contract , an activity diagram that shows the abstract idea behind the nested operations, and an interaction diagram that shows the communication between the classes of Jakamar.

3.5.3.1. Storage

Contract	
Responsibilities	Stores the specified object in the database.
Outputs	-
Preconditions	• object's class has persistence mapping
Postconditions	• if the object had no identity, it was assigned a new identity • the object was stored in the database • if the object had related objects, and the relationship cascaded over storage, the related objects were stored

Figure 3-10 shows the abstract action taken in during storing.

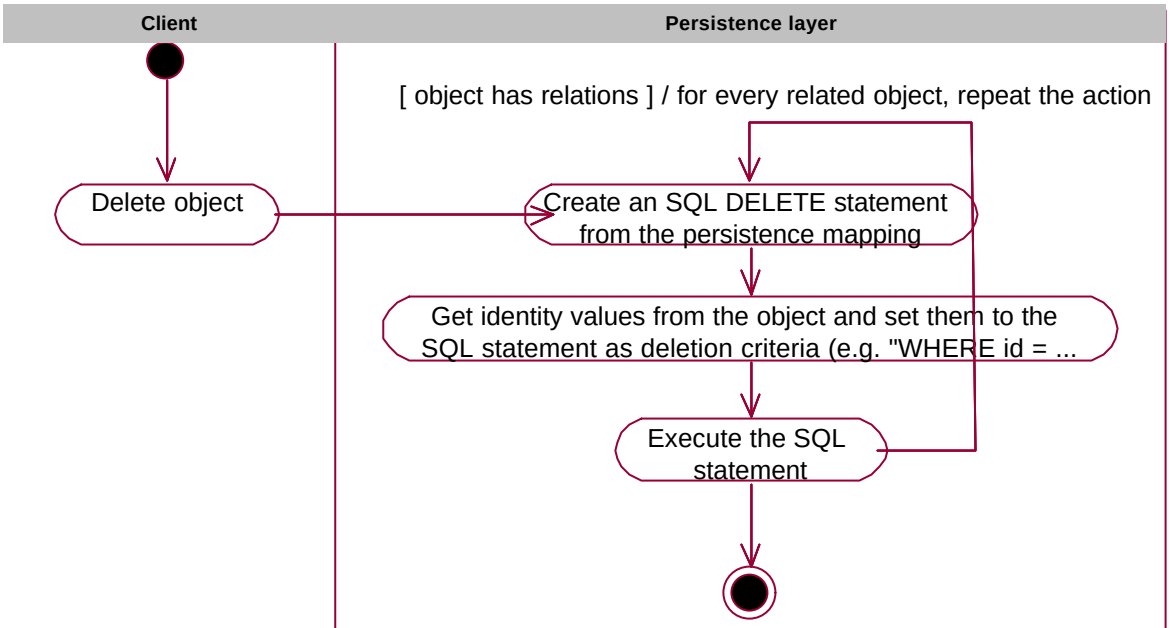

Figure 3-10. *Abstract action taken during storing an object.*

Figure 3-11 shows the concrete action taken if the object is already persistent and needs to be updated in the database.

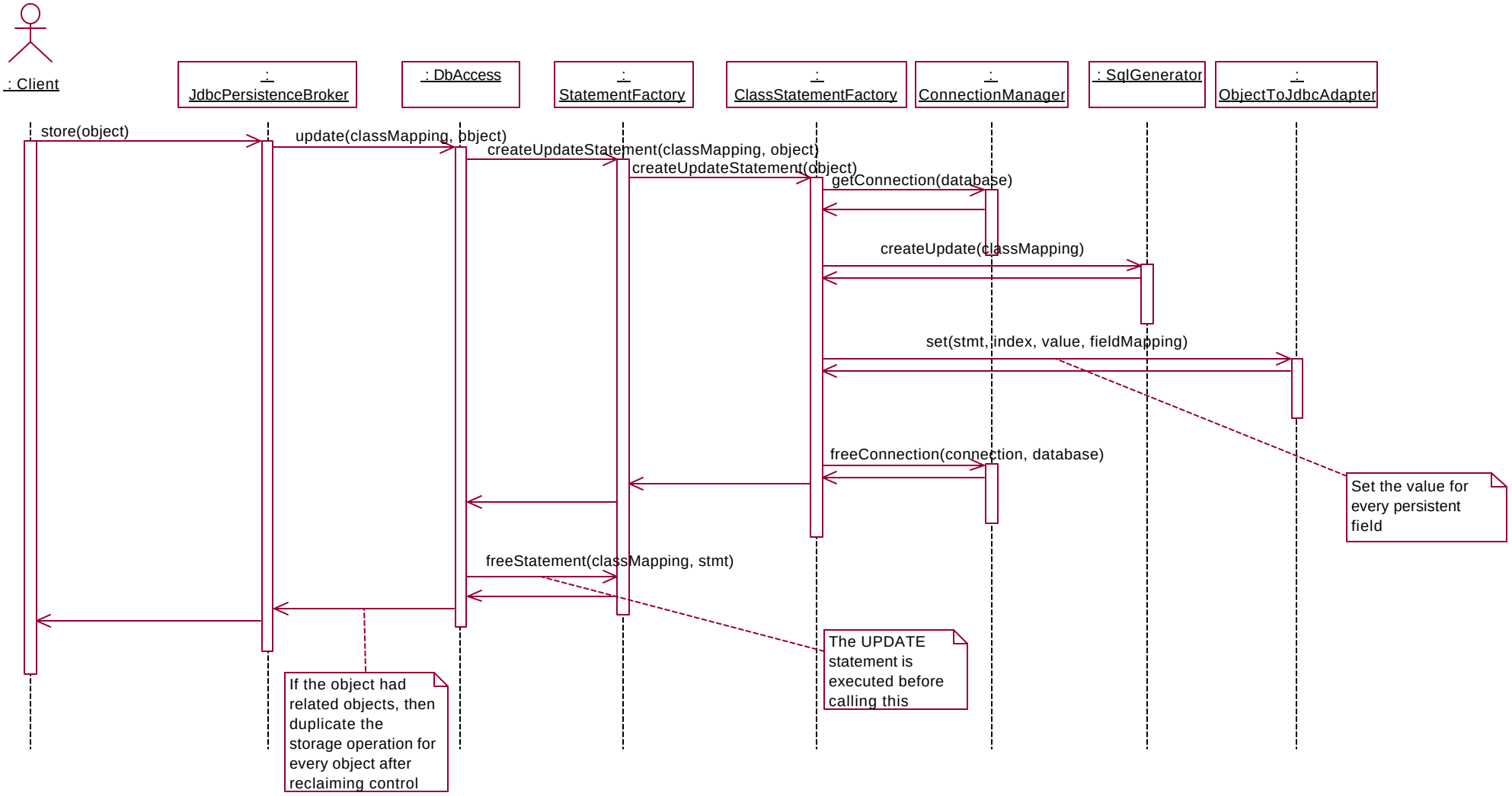

Figure 3-11. *Concrete action taken during updating an object in the database.*

Figure 3-12 shows the concrete action taken if the object has been transient so far and needs an identity before being inserted into the database.

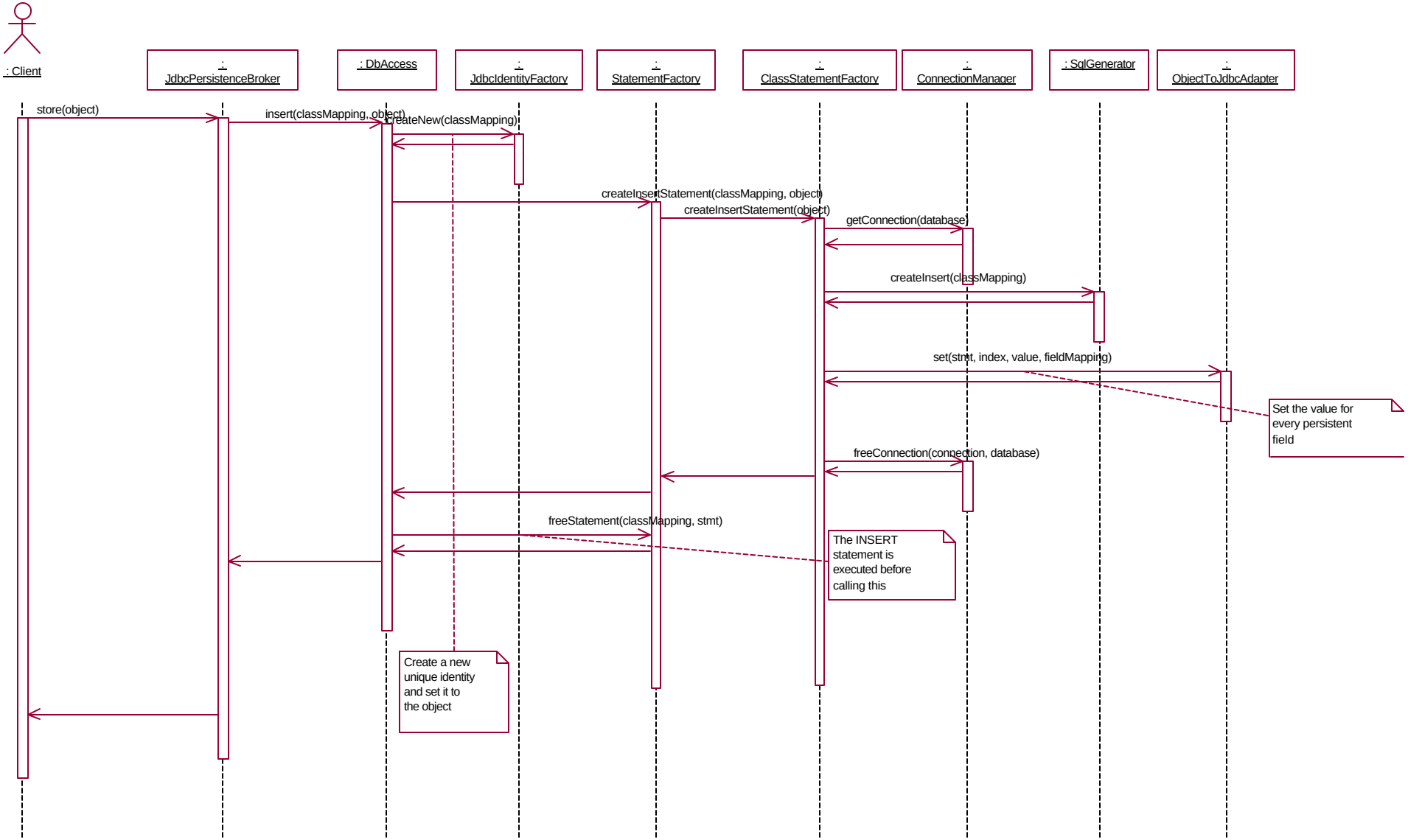

Figure 3-12. *Concrete action taken during inserting an object in the database.*

3.5.3.2. Deletion

Contract	
Responsibilities	Deletes the specified object from the database.
Outputs	-
Preconditions	• object's class has persistence mapping
Postconditions	• if the object had related objects, and the relationship cascaded over deletion, the related objects were deleted • the specified object was deleted from the database

Figure 3-13 shows the abstract action taken during object deletion.

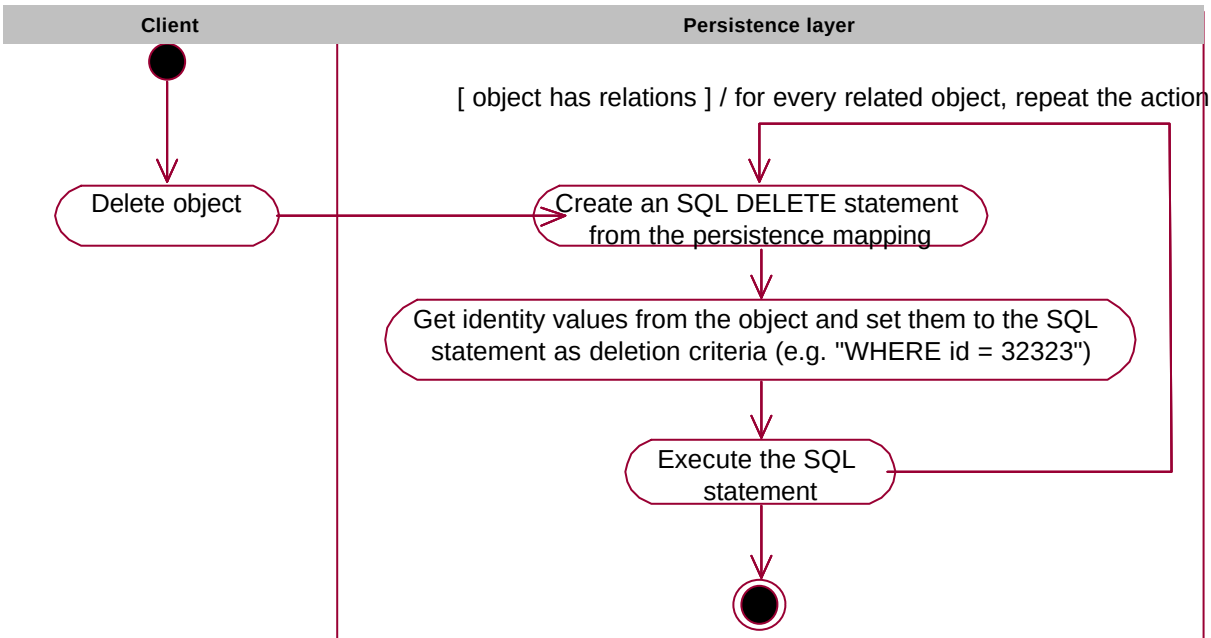

Figure 3-13. *Abstract action taken during object deletion.*

Figure 3-14 shows the concrete action taken during object deletion.

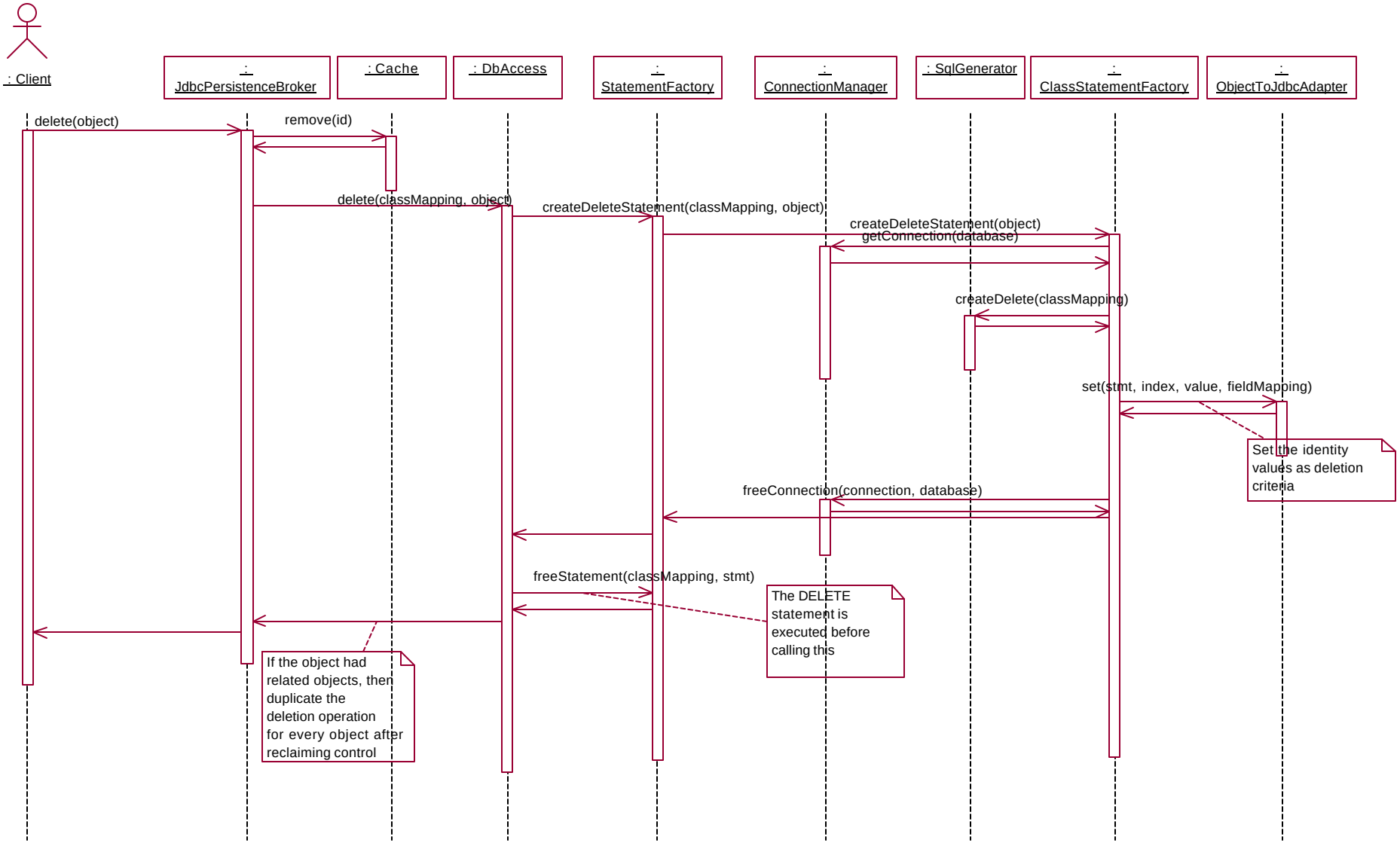

Figure 3-14. *Concrete action taken during object deletion.*

3.5.3.3. Retrieval

Contract	
Responsibilities	Retrieves objects corresponding to the specified query criteria from the database.
Outputs	retrieved objects
Preconditions	-
Postconditions	• objects corresponding to the specified criteria were retrieved from the database • if the class of the objects had a relationship with another class, and the relationship cascaded over retrieval, then the related objects were retrieved as well and set to the referencing object

Figure 3-15 shows the abstract action taken during collection retrieval.

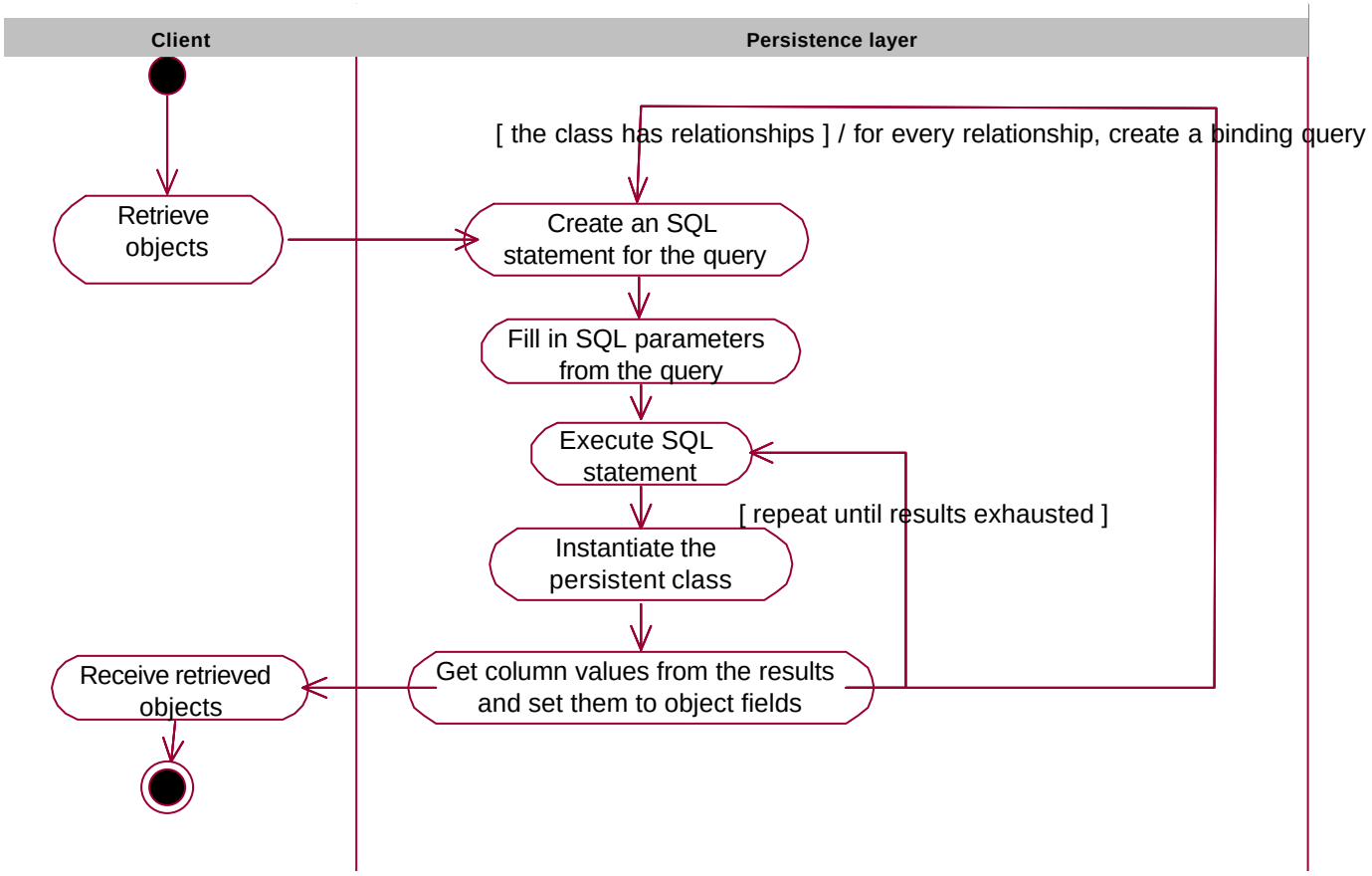

Figure 3-15. *Abstract action taken during collection retrieval.*

Figure 3-16 shows the concrete action taken during collection retrieval.

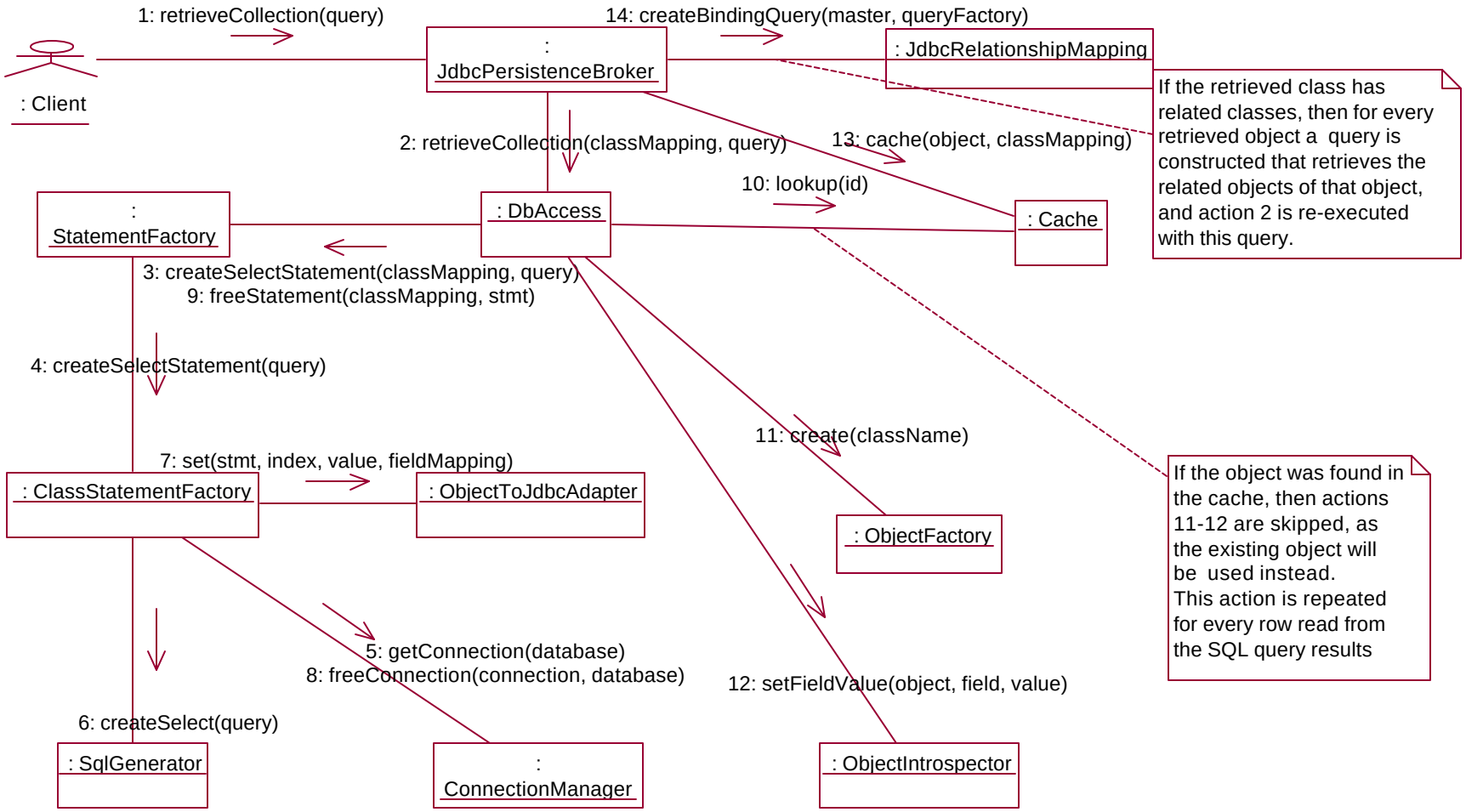

Figure 3-16. *Concrete action taken during collection retrieval.*

3.6. Error Handling

The persistence layer is quite paranoid in its regard to possible errors. All method arguments are meticulously examined for validity, even among the internal classes.

The main error handling ideology throughout the persistence layer is that error handling is *fail-fast* - that is, in case of an error, the application fails quickly and cleanly. The reasoning behind this is that most of the likely errors encountered are not transient errors (for example, data store connection failures), but rather flaws in the persistence layer configuration. Such errors need to be flushed out as quickly as possible.

However, it is possible to configure the persistence layer with an arbitrary error handling policy, with a custom implementation of the interface *jakamar.helpers.ErrorHandlingPolicy*. There are several implementations readily available in the persistence layer:

FailFastErrorHandlingPolicy

: The default error handling policy used by the persistence layer. The ideology is to fail quickly and cleanly in case of any errors.

RobustErrorHandlingPolicy

: A robust error handling policy that swallows all encountered errors, passing no errors to the client.

In addition, clients can specify an arbitrary implementation.

3.6.1. Probable Encountered Errors

Error	Details	
null as a method argument	**Time and place**	at any time, in all places
	Details	Most of the methods do not accept null as a valid argument.
	Reaction	*NullPointerException* is thrown, if the used error handling policy does not specify otherwise.
configuration file access failure	**Time and place**	creation of a *PersistenceBroker*, in *PersistenceBrokerFactory*
	Details	Configuration cannot be read from the configuration file, either because the file is not accessible, or an I/O error occurs while reading the file.
	Reaction	*ConfigurationException* is thrown.
configuration error	**Time and place**	creation of a *PersistenceBroker*, in *PersistenceBrokerFactory*
	Details	The configuration is invalid. For example, required elements are not present, or data is in invalid range.
	Reaction	*ConfigurationException* is thrown.
connection failure	**Time and place**	performing persistence operations, in *JdbcPersistenceBroker*
	Details	Attempt to connect to the database failed, or attempt to use an established connection failed.
	Reaction	*DataStoreException* is thrown, if the used error handling policy does not specify otherwise.

Error	Details	
data store error	**Time and place**	performing persistence operations, in *JdbcPersistenceBroker*
	Details	An error occurs in the database, most probably caused by invalid data, violating referential integrity or accessing locked records. Occurrence of this error most likely indicates invalid configuration, either of the persistence layer of of the database.
	Reaction	*DataStoreException* is thrown, if the used error handling policy does not specify otherwise.
reflection error	**Time and place**	performing persistence operations, in *JdbcPersistenceBroker*
	Details	An error occurs when manipulating field data via Java reflection. Can be caused by trying to set an invalid value to a field, indicating either a flawed configuration or need for a custom *ObjectToJdbcAdapter*. Another possible cause is reflection restriction. By default, field manipulation has no restrictions, but a custom *SecurityManager* may have been set to the application.
	Reaction	*PersistenceException* is thrown, if the used error handling policy does not specify otherwise.
transient object as an argument	**Time and place**	performing persistence operations, in *JdbcPersistenceBroker*
	Details	The class of the object has no persistence mapping and therefore cannot be handled by the persistence layer.
	Reaction	*ClassNotPersistenceCapableException* is thrown, if the used error handling policy does not specify otherwise.

Error	Details	
query verification error	**Time and place**	performing queries, in *JdbcPersistenceBroker*
	Details	The query cannot be compiled because of illegal settings. For example, a field to order the results by has no persistence mapping.
	Reaction	*InvalidQueryException* is thrown, if the used error handling policy does not specify otherwise.
illegal range of retrieval	**Time and place**	performing retrieval of multiple objects, in *JdbcPersistenceBroker .retrieveCollection(Query query, int offset, int count)*
	Details	The specified range of objects to return from the query results is not valid - the offset is negative, or the count is less than -1.
	Reaction	*IndexOutOfBoundsException* is thrown, if the used error handling policy does not specify otherwise.

3.7. Security

In a Java application, security must be considered from three different contexts: virtual machine security, application security and network security [Sund01].

3.7.1. Virtual Machine Security

Virtual machine security is mostly concerned with the introduction of unverified byte code (allowing unsafe or illegal code to run), and with the subversion of the Java type system (allowing to manipulate an object in ways the author never intended; accomplished by using unverified code).

By default, the JVM does not run unverified byte code. The verification can be turned off by the user, though. If there is a concern that the virtual machine could be attacked, verification should not be disabled.

3.7.2. Application Security

Application security is by far the largest security domain, concerned with the problem areas, shortcomings and flaws of design and implementation. Here the focus is laid on problem areas in application security.

3.7.2.1. Tracking Operations

Problem

The history of potentially sensitive operations must be accessible.

Assessment

Situations can arise where information has mysteriously been changed or deleted, and it is important to know when it happened and what the previous data looked like.

Solution

- Enable adequate logging in the persistence layer. See *3.9 Logging*.

3.7.2.2. Consuming Unreasonable Amounts of Resources

Problem

The persistence layer could use up too many system resources.

Assessment

The persistence layer in a newly initialized state holds only mapping information, constructed from the configuration file. Additional work objects (like statement factories for individual classes) are initialized on demand.

Database connections are also created on demand. They are not closed over the life of the application, as creating a connection is a time-costly operation. Persistent objects are cached, to speed up object retrieval. The cache uses soft references that are garbage-collected in response to a memory demand.

To conclude: as the persistence layer uses lazy initialization wherever possible to avoid unnecessary resource allocation, and keeps no objects that are not needed, memory is used in reasonable amounts. If the memory footprint is still too large, object caching can be disabled.

3.7.2.3. Confidentiality of Authentication Data and Persistent Data

Problem

The authentication information of the data store (e.g. username and password for connecting to the database) is stored in the configuration file. If the application launched by a malicious user has access to this file, or if the malicious user can access this information directly, then the authentication information is compromised, and the malicious user can steal, alter and destroy persistent data.

Assessment

This concern usually arises if the application is a web application, for example a servlet, that by default is not run with the access rights of the user and thus the configuration file must have more liberal access rights.

If no solution is used, then harmful activity on persistent data is at least detectable if adequate logging is enabled (see *3.9 Logging*).

Solutions

- The configuration file can also be a URL. Limit the access to the URL (e.g. by setting a password in the web server and providing the access information inside the configuration URL string, in the form of http://username:password@www.foobar.foo/conf.ini). This solution provides mediocre security at best - if the malicious user has access to the application code (either reading the source code directly or decompiling the Java class file) or is eavesdropping on the network, then the configuration data is still compromised.
- Set the access permission of the file to be accessed by the attacked user only, and launch the application with the rights of the attacked user.

3.7.3. Network Security

Network security is concerned with protecting data sent over the network.

3.7.3.1. Sending Sensitive Data over The Network

Problem

If access to the data store is performed over the network, then sensitive information can be obtained by eavesdropping. This includes the authentication information and persistent data. Information could also be tampered with - changed on its way through the communications channel.

Assessment

The persistence layer relies on the JDBC driver for connecting to the database. No attempt is made on the part of the persistence layer to secure the information exchange. Therefore, eavesdropping is a very real risk.

Solutions

- Use a secure JDBC driver. With a secure driver, the communication between the driver and the database is encrypted. The IDS JDBC driver is an example of a secure JDBC driver (http://www.idssoftware.com/jdbcdrv.html).
- Create a secure information exchange tunnel. Direct the database connection through the secure tunnel (e.g. using SSH port forwarding).

3.8. Concurrent Use

The persistence layer can be used in a multi-threaded environment, with multiple clients using the same *PersistenceBroker* object.

PersistenceBrokerFactory retains the persistence brokers it has constructed and returns an existing instance if one exists for the specified configuration file.

As the threads utilize the same resources, there will quite probably be some resource conflicts.

Database connection is a shared resource. If two threads want to access the database at the same time, one will either have to wait until the persistence layer has finished serving the first thread, or a pool of connections can be used.

Persistent objects are a shared resource. As retrieved objects are cached, then several threads retrieving the same object are given a reference to the same object. If the threads use the object as read-only, then no problem occurs. If, however, the threads modify the objects, then undefined behaviour can occur in the thread context.

Database records are a shared resource. If a thread modifies a record, it is locked. If another thread tries to modify the record at the same time, then the result depends on the particular database implementation. Most probably an exception will be thrown by the JDBC driver.

In case there is need for a persistence broker not shared with any other threads, the method *PersistenceBrokerFactory.createNew* can be used. This method does not use the cached persistence brokers, but creates a new one, that will not be given to other callers.

3.9. Logging

The persistence layer makes heavy use of logging. Every performed action is logged. Logging enables to debug the application, to examine unexpected behaviour and to perform auditing.

It is possible to configure logging at runtime. Logging behaviour can be controlled by editing a logging configuration file.

There are 5 levels of logging. They are ordered ascendingly and are nested - lower levels include the higher levels. The levels are:

DEBUG — Logging every operation detail. This level should be used only for debugging purposes, as the output is extremely abundant and can amount to thousands of entries in a few calls to a persistence broker instance.

INFO — Logging application progress - what operations are performed, what values stored, what data retrieved. This level should be used if there is need for auditing information. The output is nowhere near as abundant as on the previous level, but as all stored and retrieved values are logged, it can still reach undesirable quantities.

WARN — Logging potentially harmful situations that do not have affect on program flow.

ERROR — Logging error situations that interrupt the current program flow, but can be possible to recover from.

FATAL — Logging unrecoverable error situations that make it impossible for the application to continue. Setting the logging level to FATAL practically disables logging, as fatal errors are extremely rare under normal circumstances.

The logging system used is the log4j package (a popular open source logging system, available at http://jakarta.apache.org/log4j/). The target of the log output can be the console, a file, a remote socket, a remote Unix Syslog daemon, a Windows NT Event logger, a database table, an e-mail account, a JMS channel etc. Log messages can be sent to multiple targets simultaneously - for example, written to the console and to a file.

For an example logging configuration file, see *Appendix C: Sample Logging Configuration*.

3.10. Configuration

An instance of a persistence broker is created and configured from a configuration file, which is in XML format. The configuration file contains the information for mapping Java classes onto a persistence mechanism, and provides options for fine-tuning the persistence layer (like specifying the exact class of the cache).

For the Document Type Definition (DTD) of the configuration file of a *JdbcPersistenceBroker*, see *Appendix A: Configuration File Syntax*.

For an example configuration file of a *JdbcPersistenceBroker*, see *Appendix B: Sample Configuration*.

The following items deal some issues, subject to frequent concern, that can be solved through configuration. In all cases, configuring is done using pluggable interfaces - client programmer specifies the concrete class implementing the interface that Jakamar will use.

3.10.1. Column-to-Field Conversions

At times, special conversion is needed to set the value of a table column to an object field, and vice versa. For example, a common issue is that the object field is of type boolean, but the table column is of type integer.

Jakamar uses pluggable classes that act as a conversion layer between objects and tables - implementations of the *jakamar.helpers.ObjectToJdbcAdapter* interface.

To solve a special conversion issue, the client programmer must write a custom implementation of the interface and configure the persistence broker to use it for the particular class or field.

An example of configuring the persistence broker to use a custom adapter:

```
<classmapping id="Book" name="Book" database="bookdb"
table="book">
    <fieldmapping name="author" column="book_author"/>
    <fieldmapping name="title" column="book_title"/>
    <fieldmapping name="isOutOfStock"
     column="book_out_of_stock"
     adapter="foo.bar.BooleanToIntAdapter"/>
</classmapping>
```

3.10.2. Accessing Object Fields

By default, Jakamar uses reflection to inspect and mutate the values of object fields. However, it is possible that such an approach is unfit for a certain class. For example, the class needs to fill the contents of another field as well, performing some custom changes. Jakamar uses pluggable classes that act as adapters between the persistence layer and the object fields - implementations of the *jakamar.helpers.ObjectIntrospector* interface.

To solve a special access issue, the client programmer must write a custom implementation of the interface and configure the persistence broker to use it for the particular class or field.

An example of configuring the persistence broker to use a custom introspector:

```
<classmapping id="Book" name="Book" database="bookdb"
table="book" introspector="foo.bar.BookIntrospector">
    <fieldmapping name="author" column="book_author"/>
    <fieldmapping name="title" column="book_title"/>
</classmapping>
```

3.10.3. Obtaining New Instances of Persistent Classes

By default, Jakamar uses reflection to instantiate persistent classes. It is possible that a class has a more complicated instantiation progress - for example, instances are obtained from a factory class.

Jakamar uses pluggable classes that act as adapters between the persistence layer and object creation strategy - implementations of the *jakamar.helpers.ObjectFactory* interface.

To solve a special instantiation issue, the client programmer must write a custom implementation of the interface and configure the persistence broker to use it for the particular class or field.

An example of configuring the persistence broker to use a custom introspector:

```
<classmapping id="Book" name="Book" database="bookdb"
table="book" factory="foo.bar.BookFactory">
    <fieldmapping name="author" column="books_author"/>
    <fieldmapping name="title" column="books_title"/>
</classmapping>
```

3.10.4. Special SQL Syntax

Databases by different vendors can use syntax different from the one that is generated by Jakamar by default.

Jakamar uses pluggable classes that generate SQL statements for a given database - implementations of the *jakamar.jdbc.SqlGenerator* interface.

To handle a database with a different flavor of SQL, the client programmer must write a custom implementation of the interface and configure the persistence broker to use it for the particular class or field.

An example of configuring the persistence broker to use a custom SQL generator:

```
<database id="bookdb" driver="sun.jdbc.odbc.JdbcOdbcDriver"
 url="jdbc:odbc:ab" username="root" password="god"
 sqlgenerator="foo.bar.CustomSqlGenerator">
</database>
```

3.10.5. Generating Object Identities

There are a few types of identity generators available in Jakamar (see *3.3 Object Identities*). Jakamar uses pluggable classes that generate OIDs - implementations of the *jakamar.jdbc.IdentityGenerator* interface.

To handle a special identity generation strategy, the client programmer must write a custom implementation of the *jakamar.jdbc.IdentityGeneratorBuilder* interface and

configure the persistence broker to use that builder for creating a custom identity generator.

An example of configuring the persistence broker to use a custom OID generator:

```
<classmapping id="Book" name="Book" database="bookdb"
 table="book" identitygenerator="idgen2">
    <fieldmapping name="id" column="book_id"/>
    <fieldmapping name="name" column="book_name"/>
    <fieldmapping name="title" column="book_title"/>
</classmapping>

<identitygenerator id="idgen2"
 builder="foo.bar.CustomIdentityGeneratorBuilder"/>
```

On building a persistence broker, the specified identity generator builder class is instantiated, is fed data from the configuration file, and finally it returns a usable identity generator.

3.10.6. Caching Persistent Objects

Jakamar caches in memory the persistent objects it stores and retrieves. This can increase object retrieval dramatically. The default cache class uses memory-sensitive caching - if the application is short of memory, the cached objects are garbage collected. However, the cache class is pluggable. If there is need for a different caching strategy, the client programmer must write a custom implementation of the *jakamar.helpers.Cache* interface and configure the persistence broker to use. For example, another cache might set a size limit on the cache, or abandon cached instances based on the time they were last looked up.

Caching can also be disabled altogether.

An example of configuring the persistence broker to use a custom cache:

```
<cache class="foo.bar.CustomCacheImplementation"
 enabled="true" />
```

3.11. Further Development

The persistence layer is far from being a completed product. The first fully usable prototype contains the most vital functionality required - storing, retrieving, deleting. Several of the functionalities mentioned in the *Persistence Broker* pattern are yet to be implemented.

Overview of items scheduled for further development:

Development Item	Details	Priority
Cursors	Provide support for a cursor that points to a set of retrieved objects and that the user can move to the next or previous retrieved object. The cursor makes use of lazy initialization - an object is retrieved only when the cursor arrives at the object. The cursor will probably implement the *java.util.Iterator* interface (which is practically a general interface for a cursor).	primary
Transactions	Provide support for transactions - grouping persistence operations into atomic units that succeeds or fails as a whole. Transactions are very helpful in many applications, especially in the business realm (e.g. making a bank transfer consists of several operations - decreasing the amount on the transferor side and increasing on the transferee side - that must all either succeed or fail).	primary
Extent classes	Provide support for inheritance among persistent classes. A persistent class can have subclasses that hold additional data and specify additional behaviour. For example, class Person might have subclasses Employee and Client. When retrieving Persons, the persistence layer should return both employees and clients. This inheritance must somehow be implemented in the data store.	primary
Object Query Language	Provide support for the Object Query Language (OQL). OQL is an SQL-like declarative language that provides a rich environment for efficient querying of database objects, including high-level primitives for object sets and structures [ODMG1]. OQL has the advantage of simplicity and better overview, when compared to constructing queries by setting parameters and criteria via method calls.	secondary

Development Item	Details	Priority
Virtual proxies	Provide support for lazy initialization of persistent objects via virtual proxies. A virtual proxy is a proxy for another object (the real persistent subject) that materializes the subject when it is first referenced [Lar01]. As it could happen that the real object might never really be needed, this can provide a notable performance gain.	secondary
Distributed use	Provide support for distributed use, in which there is a central persistence broker server that multiple clients connect to and execute persistence operations on.	tertiary
Logging configurable in the same configuration as persistence	Allow logging configuration to be specified in the persistence configuration file.	optional
XML files as data stores	Provide support for using XML files as data storage mechanisms. Although XML files are not comparable with databases in terms of ease of use, performance and functionality, they have the benefit of simplicity - there is no need for a database management system, everything is contained in files. For smaller projects, they provide adequate performance.	under consider-ation
Graphical user interface for configuration	Provide a graphical application for producing configuration files. It should be able to generate database table structure and a configuration file from the class structure. Another approach would be to use an existing graphical tool (such do exist - Rational Rose is one) and add support for Jakamar via plugins or scripting.	lowest

4. Benefits and Tradeoffs

4.1. Jakamar Compared to Embedding SQL Directly

A simple approach to persistence logic is embedding SQL statements directly in application code. Most of the advantages of a persistence broker were discussed in the *Persistence Broker* pattern, see *2.3 Forces* to *2.6 Rationale*.

4.1.1. Sample Code

Simple Retrieval

The following two sections of code both retrieve all SimplePersons from the database that have a location of 'Saue'. When comparing the sections, we can see that using Jakamar yields a much more terse and understandable program code, not to mention with less effort.

Example of using embedded SQL for simple retrieval.

```
Connection connection = DriverManager.getConnection("jdbc:odbc:emt");
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(
  "SELECT * FROM SIMPLEPERSON WHERE location = 'Saue'");
Collection results = new ArrayList();
while (rs.next()) {
  SimplePerson object = new SimplePerson();
  object.id        = rs.getInt("id");
  object.firstName = rs.getString("firstName");
  object.lastName  = rs.getString("lastName");
  object.location  = rs.getString("location");
  object.phone     = rs.getString("phone");
  results.add(object);
}
connection.close();
```

Example of using Jakamar for simple retrieval.

```
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
SimplePerson param = new SimplePerson();
param.location = "Saue";
Query query = broker.createQuery(param, new String[]{"location"}, null);
Collection results = broker.retrieveCollection(query);
```

Cascaded Retrieval

The following two sections of code both retrieve all Persons from the database that have a location of 'Jõhvi', and also retrieve their Phone objects. Embedded SQL gets increasingly complex with relationships - for example, if Phone objects also had related objects, then a third subquery is needed, etc. But the Jakamar syntax does not differ from simple retrieval - if a relationship between two classes has been specified to cascade over retrieval, then it is handled automatically by Jakamar.

Example of using embedded SQL for cascaded retrieval.

```
Connection connection = DriverManager.getConnection("jdbc:odbc:emt");
Statement stmt = connection.createStatement();
ResultSet rs =
  stmt.executeQuery("SELECT * FROM PERSON WHERE location = 'Jõhvi'");
Collection results = new ArrayList();
while (rs.next()) {
  Person person = new Person();
  person.id        = rs.getInt("id");
  person.firstName = rs.getString("firstName");
  person.lastName  = rs.getString("lastName");
  person.location  = rs.getString("location");
  results.add(person);
  Statement stmtSub = connection.createStatement();
  ResultSet rsSub = stmtSub.executeQuery(
    "SELECT * FROM PHONE WHERE personId = " + person.id);
  while (rsSub.next()) {
    Phone phone = new Phone();
    phone.id       = rsSub.getInt("id");
    phone.personId = rsSub.getInt("personId");
    phone.phone    = rsSub.getString("phone");
    person.phones.add(phone);
  }
  stmtSub.close();
}
stmt.close();
connection.close();
```

Example of using Jakamar for cascaded retrieval.

```
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
Person param = new Person();
param.location = "Jõhvi";
Query query = broker.createQuery(param, new String[]{"location"}, null);
Collection results = broker.retrieveCollection(query);
```

Storing

The following two sections of code both store a new SimplePerson in the database. With Jakamar, the action is even more simple than retrieval..

Example of using embedded SQL for storing data.

```
Connection connection = DriverManager.getConnection("jdbc:odbc:emt");
Statement stmt = connection.createStatement();
SimplePerson person = new SimplePerson();
ResultSet rs = stmt.executeQuery(
  "SELECT MAX(id) + 1 AS newId FROM SIMPLEPERSON");
rs.next();
// The new id value is got this value to be able to set it to the
// object immediately.
person.id = rs.getInt("newId");
rs.close();
person.firstName = "John";
person.lastName  = "Doe";
person.location  = "Neverneverland";
person.phone     = "555-5656";
stmt.executeUpdate(
  "INSERT INTO SIMPLEPERSON (id,firstName,lastName,location,phone)" +
  "VALUES (" + person.id + ", '" + person.firstName + "', '" +
  person.lastName + "', '" + person.location + "', '" +
  person.phone + "')");
stmt.close();
connection.close();
```

Example of using Jakamar for storing data.

```
PersistenceBroker broker = PersistenceBrokerFactory.create("conf.xml");
SimplePerson person = new SimplePerson();
person.firstName = "John";
person.lastName  = "Doe";
person.location  = "Neverneverland";
person.phone     = "555-5656";
broker.store(person);
```

4.1.2. Tradeoffs

The initial disadvantage of choosing to implement the *Persistence Broker* pattern is that it is a non-trivial task, possibly greater than the developing the application itself. However, as the component is easily reusable in consecutive applications, it pays off.

The main disadvantage of using such a persistence broker is a decrease in the persistence functionality. With embedded SQL, programmers can execute incredibly complex queries just as easily as simple storage and retrieval. With a persistence broker, the programmer has firm boundaries on persistence functionality. Therefore, using a persistence broker is mostly suitable for applications that keep no business logic in the database and that do not need to perform complex data mining and analysis. For applications that do their work with business objects that need to be persistent, a persistence broker is an excellent solution.

Another tradeoff with using a persistence broker might be a decrease in speed. As such a component can have an intricate inner structure, persistence operations can develop a notable overhead. The following chapter explores this issue by benchmarking simple persistence operations.

4.1.3. Benchmarks

For information about the test environment, the data structure and the columns of the table of results, see *4.4 Benchmark Information*.

Action 1 - find all SimplePersons from the location 'Saue'.

Repeated?	Ops	Embedded SQL		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	450	4500	10,00	5400	12,00	1,20
Yes		1850	4,11	1500	3,33	0,81

Action 2 - update all the SimplePersons retrieved in action 1.

Repeated?	Ops	Embedded SQL		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	450	3350	7,44	3450	7,67	1,03
Yes		1650	3,67	3300	7,33	2,00

Action 3 - find all Persons from the location 'Jõhvi' and retrieve their Phones automatically.

Repeated?	Ops	Embedded SQL		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	50	8450	169,00	8750	175,00	1,04
Yes		5650	113,00	960	19,20	0,17

Action 4 - insert a number of new Persons and two Phones for each person.

Repeated?	Ops	Embedded SQL		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	50	2200	44,00	4850	97,00	2,20

When examining the results, we can see that Jakamar's main strength lies in retrieval - in some cases it can be very fast, e.g. when executing repeated retrievals in action 3. This good performance is caused by caching - as objects that are cached do not need to be fully read from the database (once their identity is clear, their cached instance can be reused), several operations can be skipped altogether. An especially high performance yield comes with relationships - cached objects already have their related objects set in place, so additional queries to the database for getting the related objects are not needed.

On the whole, the results were surprisingly good. I did not expect all the application logic existent in Jakamar to have such a small time overhead, and was fearing at least a triple time penalty with using Jakamar.

On the basis of these results, using a persistence broker does not make persistence operations tremendously slower, but as seen from the code samples, yields much less code to develop.

4.2. Jakamar Compared to Similar Components

There is abundant software available that performs mapping Java objects to relational databases. Some products are quite powerful, providing a rich user interface for configuration and offering automatic generation of the class diagram from the database table diagram. However, most products do not have the level of unintrusiveness that the *Persistence Broker* pattern, and consequently the Jakamar persistence layer, have. Persistent classes usually have to extend a superclass that couples them with the persistence layer.

There is only one component that does compare to Jakamar in terms of unintrusiveness and functionality - ObJectRelationalBridge.

4.2.1. ObJectRelationalBridge[1]

ObJectRelationalBridge (OJB) is an open source software (publicly available at http://objectbridge.sourceforge.net/) roughly following the *Persistence Broker* pattern. Its functionality and structure is quite similar to Jakamar.

Advantages compared to Jakamar

- supports cursors (for information on cursors, see *3.11 Further Development*)
- has partial support for transactions - only if the underlying database supports transactions. Some databases do not support transactions, like MySQL (http://www.mysql.com/).
- supports virtual proxies (for information on virtual proxies, see *3.11 Further Development*)
- supports extent classes (for information on extent classes, see *3.11 Further Development*)

[1] By February 2002, a new version of OJB has cleared several of the concerns mentioned here, and the benchmarks are suspect as well. The following should be read with that in mind.

Disadvantages compared to Jakamar

- Has virtually no logging. It is possible to have some messages printed to the console after recompiling the classes to support this, but the messages are inexpressive and few.
- Error handling is severely lacking - in most cases, encountered exceptions are caught and not let to rise outside the component, thus giving no clue if there was an error.
- Does not support composite identities.
- Table columns can only have certain pre-defined types - the types are hard-coded in. Although they cover most of the possible column types, they do not cover all.
- Object identity fields are hard-coded to integer type.
- There is only one identity generation strategy - using a sequence table in the database. It is possible to configure OJB to use an arbitrary strategy, but not on per-class basis or even a per-persistence broker basis - the specified strategy will then be used by all persistence brokers. Configuring this arbitrary strategy implementation has to be done outside the application - the broker configuration file holds no information for it.
- There is one global object cache for the entire machine. As caches do not know where an object is from - they only know its type and identity values, this can lead to problems in cases where there are multiple applications running on the same machine that use the same persistent classes, but store them in different data stores.
- Contains some bugs - for example, one type of query disregards its criteria and returns every object for the specified class it can find.
- Instantiating persistent classes is not flexible - the user cannot specify an arbitrary object creation strategy.
- Does not support custom SQL.
- Does not support a custom way for accessing object fields - the access is hard-coded to be done using reflection and cannot be changed.
- API documentation is inadequate, incomplete and defective.

4.2.2. Benchmarks

For information about the test environment, the data structure and the table columns, see *4.4 Benchmark Information*.

Action 1 - find all SimplePersons from the location 'Saue'.

Repeated?	Ops	OJB		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	450	6550	14,56	5400	12,00	0,82
Yes		3200	7,11	1500	3,33	0,47

Action 2 - update all the SimplePersons retrieved in action 1.

Repeated?	Ops	OJB		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	450	10750	23,89	3450	7,67	0,32
Yes		10350	23,00	3300	7,33	0,32

Action 3 - find all Persons from the location 'Jõhvi' and retrieve their Phones automatically.

Repeated?	Ops	OJB		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	50	14550	291,00	8750	175,00	0,60
Yes		1650	33,00	960	19,20	0,58

Action 4 - insert a number of new Persons and two Phones for each person.

Repeated?	Ops	OJB		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	50	10500	210,00	4850	97,00	0,46

These benchmark results are even more surprising than those in comparison with embedded SQL. I expected OJB, a product that has had a considerably longer lifetime, to be much better in terms of performance than Jakamar, a product that is a newborn.

It is possible that such good performance is caused by some issues that need to be handled but have not occurred to me.

On the basis of these results, Jakamar can be pronounced to give excellent performance.

4.3. Conclusion

The *Persistence Broker* pattern is a good approach to handling data persistence. As we saw from its comparison with directly embedded SQL, it does not incur an unacceptable performance . In many cases, a persistence broker can give results amazingly fast, because it caches the persistent objects it encountered.

However, the main argument for a persistence broker is its ease of use. Utilizing it delivers clean and elegant application code, faster development time (as the programmer does not need to produce any persistence logic), easy persistence maintenance (changes are needed only in the configuration file) and good portability.

On the basis of these results, we can conclude that the *Persistence Broker* is a good approach to handling data persistence, although not a solution fit for every application.

When comparing Jakamar, one implementation of the *Persistence Broker* pattern, to another implementation, OJB, we can see that the main strength of Jakamar is in its careful design and flexibility - it has powerful logging, good error handling and its architecture is compartmentalized so that custom subcomponents can be used with the basic aspects of persistence operations, like creating SQL syntax and accessing object fields.

For the time being, Jakamar is lacking some useful functionality that OJB possesses, like supporting cursors and transactions. However, these items are already on the list for future developments.

The benchmarks showed that Jakamar has surprisingly good performance compared with OJB. It is difficult to say what causes this good performance, as OJB does not log its activity and inspecting source files and profiling the performance of every single operation is very time-consuming.

On the basis of the comparison and the benchmarks, Jakamar can be pronounced viable and quite promising. When future development cycles have filled the present voids in Jakamar's functionality, it could become a widely-used component.

4.4. Benchmark Information

The logging inside Jakamar was disabled during benchmarking in order to concentrate on the performance of the persistence operations.

Test environment

Testing was carried out on a machine that hosted both the application and the database server.

Computer:	Celeron 433 MHz, 128 MB RAM, Windows 95
Database system:	Microsoft Access 97
Database driver:	Sun's JDBC-ODBC bridge

Test Data Structure

Table SIMPLEPERSON contained 90,000 records.

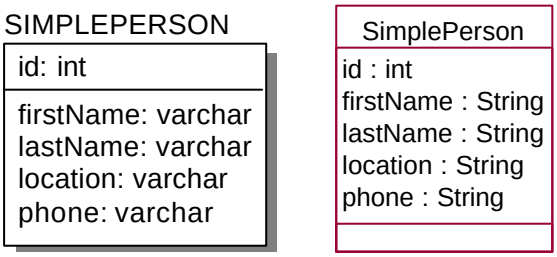

Figure 4-1. *Relational and object-oriented data structure for SimplePerson*

Table PERSON contained 6,800 records. Table NUMBER contained 16,500 records and had a foreign key column referencing the PERSON table.

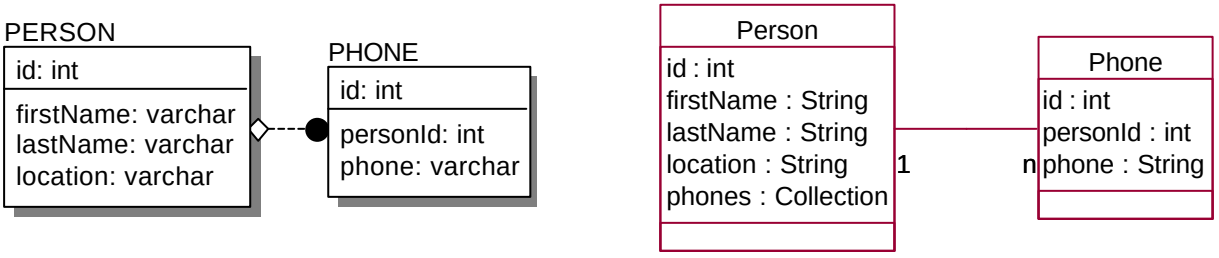

Figure 4-2. *Relational and object-oriented data structure for Person and Phone.*

The following operations were benchmarked:

1. retrieving all SimplePersons for the location 'Saue'
2. updating every entry among the results retrieved in the above operation
3. finding all Persons from the location 'Jõhvi' and retrieving their Phones automatically
4. inserting a number of new Persons and two Phones for each person

Results Table

Repeated	whether the same operation was repeated for multiple times.
Ops	how many atomic operations - storings or retrievals - the action contained. For example, action 1 retrieved 450 SimplePerson objects.
Sum	how much time did all the ops take.
Avg	how much time did an op take on the average.
Factor to Jakamar	by what factor is X faster that Jakamar (if below 1, then Jakamar is faster than X)

Repeated?	Ops	X		Jakamar		Factor to Jakamar
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	

Note that I carried out the benchmarking personally. It is generally recommendable to let a neutral third party perform this action, because as someone who is deeply familiar with Jakamar, my testing might have been unconsciously tuned in favor of Jakamar. However, due to limited time, I had no chance to obtain outside help.

5. Conclusion

I set out to achieve the following goals:

- to propose a reusable generic approach to handling persistence logic - the *Persistence Broker* design pattern, that solves several problems when using relational database management systems in an object-oriented application environment. The basic idea for such software was introduced in Scott W. Ambler's whitepaper "Mapping Objects to Relational Databases" [Ambl00a].
- to present Jakamar, a component I had written in the Java programming language as an implementation of this pattern
- to compare Jakamar's performance to the most basic approach to persistence logic to find out whether the *Persistence Broker* offers a more preferable solution; the most basic approach being embedding database access logic directly into application code. Jakamar was also compared to other software products that follow the *Persistence Broker* pattern, to find out whether Jakamar is a viable solution among similar components.

All the three goals were realized. Based on the results of the comparison with the basic approach, I deemed the *Persistence Broker* to be a well-designed approach to persistence logic. Using Jakamar yielded faster program development, better modularized application structure, and in cases even better performance on persistence operations.

The software market has very few software products available that follow the *Persistence Broker* concept strictly. Among the similar available software, Jakamar performed very well. Its functionality could be more extended, but the missing functionality was already planned into future development.

6. Glossary

API	stands for Applications Programming Interface. It is the interface to a collection of classes that tells what operations are available and what do they do, but usually discloses their implementation details
composite identity	an *object identity* that has a value composed of several fields
enterprise application	a combination of building block components that work together to perform a business function, usually accessible over the network [IBMWS1]
garbage collection	identifying no-longer used memory which was dynamically allocated and returning it to a pool of free memory [Mar97]
introspection	see *reflection*
JDBC	the Java API for executing SQL statements, consisting of a set of classes and interfaces written in the Java programming language [Grah97]
JMS	the Java Message Service, a specification for message passing and related operations among distributed software components [JMS01]
JVM	the Java virtual machine. An abstract computing machine that executes Java programs.
lazy initialization	the program refrains from creating certain resources until the resource is first needed, thus keeping resource consumption at a valuable minimum [Bis1]
materialization	the process of retrieving a persistent object from a data store
object database	(*alias* object-oriented database, object-oriented database management system) a database management system that supports the modelling and creation of data as objects

object identity	(*alias* OID) an identifier that, assigned to a persistent object, uniquely identifies the object
OID	see *object identity*
persistence	the ability of objects to exist beyond the lifetime of the application [Bal1]
persistence layer	a collection of classes that provide objects the ability to be persistent, being effectively a wrapper for the persistence mechanism [Ambl00b]
persistence logic	application code that handles storing, deleting and retrieving *persistent data*
persistent data	data that has *persistence*
persistence operarions	operations like saving, deleting and retrieving persistent data
pluggable interface	denotes a situation where an instance of a specific interface can be "plugged in" to the application. The application accepts an instance of the interface and uses the interface operations, without knowing or caring about the concrete class that is providing the implementation behind the operations.
reflection	the process of inspecting a class for meta-information - information about its fields and methods, calling methods and accessing fields dynamically, etc. Also known as *introspection*. [McM97]
servlet	a Java class that has the functionality of a w*eb application* - it can be called over the web and it responds by sending back content
soft reference	an object reference that is cleared at the discretion of the garbage collector in response to memory demand
transient object	object that only resides in memory and does not persist outside of an application [Sun1]

transparent persistence	automatically provided persistence without special effort on the client programmer side. No difference between persistent and transient objects. [Sun1] [Trans1]
web application	software that is installed on a web server and accessed over the network.The most common web application is an HTML document that has dynamically (on-request) generated content, e.g. a weather forecast page.

7. Kokkuvõte

Minu töö tutvustab ühte konkreetset lahendust objektide püsivuse saavutamiseks Java keskkonnas - objektide püsivuse kihti Jakamar.

Püsivad objektid on objektid, mis säilivad programmi ühest jooksmisest kauem. Tavaliselt saavutatakse see objektide salvestamisega mingisugusesse andmehoidlasse, kõige üldlevinumal juhul relatsioonilisse andmebaasi. Tarkvara üheks põhieesmärgiks on püsivate objektide tekitamine, muutmine ja vaatamine. Programmeerijatel tuleb lahendada mõningaid ikka ja jälle korduvaid probleeme, mis kerkivad esile relatsiooniliste andmebaaside kasutamisega püsivusmehhanismina objekt-orienteeritud tarkvara-keskkonnas. Suurim probleem on andmestruktuuri muudatuste sisseviimine. Juba väike muudatus võib tuua kaasa muudatuste ahelreaktsiooni programmi püsivusloogikas. Suuremad muudatused - näiteks andmebaasi vahetamine teise tootja andmebaasi vastu - on võimelised tekitama väga palju lisatööd. Teine mureküsimus on programmi disain - paljude programmide püsivusloogika on väga sarnane, erinedes põhiliselt ainult andmete täpses struktuuris. Seda sarnasust võiks ära kasutada ja luua lahendus, mis automatiseerib programmeerija tööd.

Oma töös seadsin ma järgmised eesmärgid:

- Esitada *Persistence Broker* disainimuster kui üldine korduvkasutatav püsivusloogika mehhanism. *Persistence Broker*-lahenduse korral on äriloogika püsivusloogikast eraldatud tasemeni, kus püsivad objektid ei ole teadlikud sellest, et neid salvestatakse ja taastatakse andmebaasist. Lahendust on kerge kohandada erinevate programmide andmestruktuuriga.
- Tutvustada komponenti Jakamar, mille ma kirjutasin Java keeles kui selle disainimustri realisatsiooni. Jakamar on objektide püsivuse kiht, mille ainsaks eesmärgiks on olla ehituskiviks teiste programmide arendamisel. Jakamari puhul on püsivusmehhanism täielikult automatiseeritud - programmeerijal pole vajadust kirjutada ühtegi rida andmebaasiga suhtlemisest, andmete salvestamise, kustutamise ja taastamise eest hoolitseb komponent ise.

- Võrrelda Jakamari kasutust tavajuhuga, mille puhul andmebaasiga suhtlemise loogika sisaldub programmis endas, ja teiste tarkvaratoodetega, mis järgivad *Persistence Broker* disainimustrit.

Kõik kolm eesmärki täideti. Ma jõudsin järeldusele, et *Persistence Broker*-i puhul on püsivusloogika hästi kujundatud, ja et Jakamar on elujõuline tarkvarakomponent. Jakamar koondas endasse terve programmi püsivusloogika ja võimaldas juurdepääsu andmete püsivusele lihtsate toimingute kaudu, nagu *salvesta(objekt)*, *kustuta(objekt)* ja *taasta(kriteeriumid)*. Lisaks eespool kirjeldatud probleemide lahendamisele andis Jakamari kasutamine paremini liigendatud programmistruktuuri, kiirema programmiarenduse (kuna programmeerija ei pea ise implementeerima andmebaasiga suhtlemist) ja võimaluse objektide taastamise kiiremaks teostamiseks.

8. References

[AGCS1] AG Communications Systems. *AG Communication Systems Pattern Template.* http://www.agcs.com/supportv2/techpapers/patterns/template.htm.

[Ambl00a] Scott W. Ambler (October 2000). *Mapping Objects To Relational Databases*. http://www.AmbySoft.com/mappingObjects.pdf.

[Ambl00b] Scott W. Ambler (November 2000). *The Design of a Robust Persistence Layer For Relational Databases*. http://www.ambysoft.com/persistenceLayer.pdf.

[Bal1] Konda R. Balabbigari. *Java Object Serialization*. JavaCaps, http://www.javacaps.com/java_serial.html.

[BibTec01] Biblio Tech Review (April 2001). *Database Management Systems (DBMS)*. Biblio Tech Review, Technical Briefings, http://www.biblio-tech.com/html/databases.html.

[Bis1] Philip Bishop and Nigel Warren. *Lazy instantiation*. JavaWorld, http://www.javaworld.com/javaworld/javatips/jw-javatip67.html.

[Gam95] Erich Gamma et al (1995). *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Grah97] Graham Hamilton et al (June 1997). *JDBC Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley.

[IBMWS1] IBM WebSphere InfoCenter. *What are enterprise applications?*. http://www-4.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/was/0001.html.

[JMS01] Sun Microsystems (October 2001). *Java™ Message Service API.* http://java.sun.com/products/jms/index.html.

[Lar01] Craig Larman (July 2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Addison-Wesley.

[Mar97] Vern Martin (December 1997). *Garbage Collection In Java.* http://trident.mcs.kent.edu/~vmartin/proj/proj.html.

[McM97] Chuck McManis (September 1997). *Take an in-depth look at the Java Reflection API.* JavaWorld, http://www.javaworld.com/javaworld/jw-09-1997/jw-09-indepth.html.

[ODMG1] Object Data Management Group. *ODMG Standard Overview*. http://www.odmg.org/standard/standardoverview.htm.

[Sun1] Sun Microsystems. *Transparent Persistence*, Java Application Developer Central, http://www.jadcentral.com/newscentral/feature.jsp?feature_ID=8.

[Sund01] Todd Sundsted (June 2001). *Secure your Java apps from end to end, Part 1*. JavaWorld, http://www.javaworld.com/javaworld/jw-06-2001/jw-0615-howto.html.

[Trans1] *Transparent persistence in object-relational mapping*, http://www.object-relational.com/articles/transparent_persistence.html.

[UML97] Rational Software Corporation (January 1997). *UML Semantics*.

9. Appendices

9.1. Appendix A: Configuration File Syntax

Document Type Definition (DTD) for the configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  The order in which the elements appear is not relevant, save for
  identitygenerators - they have to appear after tables and
  classmappings.
-->

<!--
  The attribute "builder" specifies the full class name of the
  jakamar.PersistenceBrokerBuilder implementation to be used for
  building this PersistenceBroker. This attribute is mandatory.
  The attribute "errorhandlingpolicy" specifies the full class name
  of the jakamar.helpers.ErrorHandlingPolicy implementation to be used
  as the error handling policy in the built persistence broker. This
  attribute is not  mandatory and defaults to
  "jakamar.helpers.FailFastErrorHandlingPolicy".
-->
<!ELEMENT persistenceconfiguration (cache?, database+, classmapping+,
identitygenerator* ) >
<!ATTLIST persistenceconfiguration
  builder CDATA #REQUIRED
  errorhandlingpolicy CDATA #IMPLIED
>

<!--
  The configuration of the internal object cache of the persistence
  framework. This element is not mandatory.

  The attribute "class" specifies the full class name of the
  jakamar.Cache implementation used. This attribute is not mandatory
  and defaults to "jakamar.DefaultCache".
  The attribute "isenabled" specifies whether the cache is enabled.
  If not true, then this setting overrides all other caching
  settings in classmappings. Enabling the cache speeds up object
  retrieval. This attribute is not mandatory and defaults to "true".
-->
<!ELEMENT cache EMPTY >
<!ATTLIST cache
  class CDATA #IMPLIED
  enabled ( true | false ) "true"
>

<!--
  Information about a database. At least one instance of this element
  is mandatory.
```

```
  The attribute "id" specifies the id of the database. This attribute
  is mandatory.
  The attribute "driver" specifies the full name of the JDBC driver,
  e.g. "sun.jdbc.odbc.JdbcOdbcDriver". This attribute is mandatory.
  The attribute "url" specifies the URL of the database, e.g.
  "jdbc:odbc:testdb". This attribute is mandatory.
  The attribute "sqlgenerator" specifies the full name of the
  jakamar.jdbc.SqlGenerator implementation to use for creating SQL for
  this database. This attribute is not mandatory and defaults to
  "jakamar.jdbc.DefaultSqlGenerator".
  The attribute "username" specifies the username for connecting to the
  database. This attribute is mandatory.
  The attribute "password" specifies the password for connecting to the
  database. This attribute is mandatory.
-->
<!ELEMENT database (table+) >
<!ATTLIST database
  id ID #REQUIRED
  driver CDATA #REQUIRED
  url CDATA #REQUIRED
  sqlgenerator CDATA #IMPLIED
  username CDATA #IMPLIED
  password CDATA #IMPLIED
>

<!--
  Information about a database table.

  The attribute "name" specifies the name of table, e.g. COURSE.
  This attribute is mandatory.
  The attribute "schema" specifies the schema of table, e.g.
  COURSE_INFO. This attribute is not mandatory.
-->
<!ELEMENT table (column+) >
<!ATTLIST table
  name ID #IMPLIED
  schema CDATA #REQUIRED
>

<!--
  Information about a database table column.

  The attribute "name" specifies the name of column, e.g. NAME. This
  attribute is mandatory.
  The attribute "type" specifies the type of the column. It must be
  the name of a constant defined in java.sql.Types, e.g. VARCHAR.
  This attribute is mandatory.
  The attribute "isprimarykey" specifies whether the column is a primary
  key column. This attribute is not mandatory and defaults to "false".
-->
<!ELEMENT column EMPTY >
<!ATTLIST column
  name ID #REQUIRED
  type ( ARRAY | BIGINT | BINARY | BIT | BLOB | CHAR | CLOB | DATE |
DECIMAL | DISTINCT | DOUBLE | FLOAT | INTEGER | JAVA_OBJECT |
LONGVARBINARY | LONGVARCHAR | NULL | NUMERIC | OTHER | REAL | REF |
SMALLINT | STRUCT | TIME | TIMESTAMP | TINYINT | VARBINARY | VARCHAR )
```

```
#REQUIRED
  isprimarykey ( true | false ) "false"
>

<!--
  Information for mapping a Java class onto a database table.

  The attribute "name" specifies the full name of the class, e.g.
  jakamar.test.Course. This attribute is mandatory.
  The attribute "database" specifies the id of the database that
  contains the table the class maps onto, described in this file. This
  attribute is mandatory.
  The attribute "table" specifies the id of the database table that the
  class maps onto, described in this file. This attribute is mandatory.
  The attribute "factory" specifies the full name of the
  jakamar.meta.ObjectFactory implementation to use for constructing
  instances of the class. This attribute is not mandatory and defaults
  to "jakamar.meta.DefaultObjectFactory".
  The attribute "introspector" specifies the full name of the
  jakamar.helpers.ObjectIntrospector implementation to use for accessing
  the fields of an object of the class. This attribute is not mandatory
  and defaults to "jakamar.helpers.DefaultObjectIntrospector".
  The attribute "iscached" specifies whether instances of this class
  should be cached by the persistence framework to speed up object
  retrieval. This attribute is not mandatory and defaults to "true".
  The attribute "adapter" specifies the full class name of the
  jakamar.jdbc.ObjectToJdbcAdapter implementation to use for
  communicating with the JDBC storage and retrieval interfaces. The
  specified adapter will be used for all the mapped fields that do not
  have their own adapter specified. This attribute is not mandatory and
  defaults to "jakamar.jdbc.DefaultObjectToJdbcAdapter".
  The attribute "identitygenerator" specifies the id of the identity
  generator used for generating new identities for the mapped class.
  This attribute is not mandatory.
-->
<!ELEMENT classmapping ( fieldmapping+, relationshipmapping* ) >
<!ATTLIST classmapping
  name ID #REQUIRED
  database IDREF #REQUIRED
  table IDREF #REQUIRED
  factory CDATA #IMPLIED
  introspector CDATA #IMPLIED
  iscached CDATA #IMPLIED
  adapter CDATA #IMPLIED
  identitygenerator IDREF #IMPLIED
>

<!--
  Information for mapping a Java class field onto a database table
  column.

  The attribute "name" specifies the name of the Java class field.
  This attribute is mandatory.
  The attribute "column" specifies the id of the database table
  column, defined in this file. This attribute is mandatory.
  The attribute "adapter" specifies the full class name of the
  jakamar.jdbc.ObjectToJdbcAdapter implementation to use for
```

```
  communicating with the JDBC storage and retrieval interfaces. This
  attribute is not mandatory and defaults to
  "jakamar.jdbc.DefaultObjectToJdbcAdapter".
-->
<!ELEMENT fieldmapping EMPTY >
<!ATTLIST fieldmapping
  name ID #REQUIRED
  column IDREF #REQUIRED
  adapter CDATA #IMPLIED
>

<!--
  Information for mapping a relationship between two classes.

  The attribute "cardinality" specifies the cardinality on the detail
  side of the relationship. A valid value is any positive integer. A
  value of 1 indicates a one-to-one relationship, a value greater than
  1 indicates a one-to-many relationship. This attribute is not
  mandatory and defaults to "1".
  The attribute "relatedclass" specifies the id of the classmapping
  for the detail class, defined in this file. This attribute is
  mandatory.
  The attribute "masterfield" specifies the name of the reference field
  in the master class; it is either a simple object reference, a Java
  array or a subtype of java.util.Collection. It is probably not a
  field with a mapping. This attribute is mandatory.
  The attribute "collectionclass" specifies the concrete implementation
  of the java.util.Collection interface to instantiate and set to the
  master field if the relationship is a one-to-many relationship, the
  field is null and is not a Java array. This attribute is not mandatory
  and defaults to "java.util.ArrayList".
  The attribute "isstorecascaded" specifies whether to automatically
  store the related object(s) with the parent. This attribute is not
  mandatory and defaults to "false".
  The attribute "isdeletecascaded" specifies whether to automatically
  delete the related object(s) with the parent. This attribute is not
  mandatory and defaults to "false".
  The attribute "isretrievecascaded" specifies whether to automatically
  retrieve the related object(s) with the parent. This attribute is not
  mandatory and defaults to "false".
  The attribute "bindings" specifies the bindings of fields between the
  master class and the detail class. The bindings are comma-separated
  and have the following format:
  "masterField = detailField, masterField = detailField".
  This attribute is not mandatory.
-->
<!ELEMENT relationshipmapping EMPTY >
<!ATTLIST relationshipmapping
  cardinality CDATA #IMPLIED
  relatedclass CDATA #REQUIRED
  masterfield CDATA #REQUIRED
  collectionclass CDATA #IMPLIED
  isstorecascaded ( true | false ) "false"
  isdeletecascaded ( true | false ) "false"
  isretrievecascaded ( true | false ) "false"
  bindings CDATA #IMPLIED
>
```

```
<!--
  The configuration of an identity generator.

  The attribute "id" specifies the id of the identity generator. If the
  id is "default", then it will be used as the default identity
  generator for those class mappings that do not have a specific
  generator set. This attribute is mandatory.
  The attribute "builder" specifies the full name of the
  jakamar.jdbc.JdbcIdentityGeneratorBuilder implementation used for
  building an identity generator. The builder approach is necessary
  because different identity generators have totally different
  configuration settings that cannot be anticipated. This attribute
  is mandatory.
-->
<!ELEMENT identitygenerator EMPTY >
<!ATTLIST identitygenerator
  id ID #REQUIRED
  builder CDATA #REQUIRED
>
```

9.2. Appendix B: Sample Configuration

A sample configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE persistenceconfiguration
 SYSTEM "persistenceconfiguration.dtd">
<persistenceconfiguration
 builder="jakamar.jdbc.JdbcPersistenceBrokerBuilder">

  <database id="books" driver="sun.jdbc.odbc.JdbcOdbcDriver"
   url="jdbc:odbc:books" username="root" password="god">
    <table name="book">
      <column name="book_id" type="INTEGER" isprimarykey="true"/>
      <column name="book_name" type="VARCHAR"/>
      <column name="book_title" type="VARCHAR"/>
      <column name="book_synapsis" type="CLOB"/>
      <column name="book_sample_chapter" type="CLOB"/>
      <column name="book_publisher_id" type="INTEGER"/>
    </table>

    <table name="publisher">
      <column name="publisher_id" type="INTEGER" isprimarykey="true"/>
      <column name="publisher_name" type="VARCHAR"/>
      <column name="publisher_www" type="VARCHAR"/>
      <column name="publisher_email" type="VARCHAR"/>
    </table>
  </database>

  <classmapping name="Book" database="books" table="book">
    <fieldmapping name="id" column="book_id"/>
    <fieldmapping name="name" column="book_name"/>
    <fieldmapping name="title" column="book_title"/>
    <fieldmapping name="synapsis" column="book_synapsis"/>
    <fieldmapping name="sampleChapter" column="book_sample_chapter"
     adapter="StringBufferToClobAdapter"/>
    <fieldmapping name="publisherId" column="book_publisher_id"/>
    <relationshipmapping cardinality="1" relatedclass="Publisher"
     masterfield="publisher" isstorecascaded="true"
     isdeletecascaded="false" isretrievecascaded="true"
     bindings="publisherId = id"/>
  </classmapping>

  <classmapping name="Publisher" database="books" table="publisher">
    <fieldmapping name="id" column="publisher_id"/>
    <fieldmapping name="name" column="publisher_name"/>
    <fieldmapping name="www" column="publisher_www"/>
    <fieldmapping name="email" column="publisher_email"/>
  </classmapping>

  <identitygenerator id="default"
   builder="jakamar.jdbc.IdentityGeneratorFromSelectMaxBuilder"/>

</persistenceconfiguration>
```

9.3. Appendix C: Sample Logging Configuration

Sample logging configuration file.

```
# Set root logger priority to DEBUG and its only appender to A1.
log4j.rootCategory=DEBUG, A1
# Set the logger priority for the jakamar.jdbc package to INFO
# and its only appender to A1.
log4j.category.jakamar.jdbc=INFO, A2

# A1 is set to be a RollingFileAppender that maintains several
# backup files from recent history
log4j.appender.A1=org.apache.log4j.RollingFileAppender
log4j.appender.A1.File=test.log
log4j.appender.A1.MaxFileSize=5MB
log4j.appender.A1.MaxBackupIndex=10

# The other logging appender, A2, writes to the console.
log4j.appender.A2=org.apache.log4j.FileAppender
log4j.appender.A1.File=System.out

# Specify the format of logging messages.
# Example output:
# 17:17:14 DEBUG jakamar.PersistenceBrokerFactory.create:133 - "d.xml"
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{HH:mm:ss} %-5p %C.%M:%L -
%m%n

# Example output:
# 2001-11-12 20:26:01 DEBUG jakamar.PersistenceBrokerFactory - "d.xml"
log4j.appender.A2.layout=org.apache.log4j.PatternLayout
log4j.appender.A2.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %-5p
%c - %m%n
```