The Design of a Robust Persistence Layer For Relational Databases

Scott W. Ambler
Senior Consultant, Ambysoft Inc.
www.ambysoft.com/scottAmbler.html

http://www.ambysoft.com/downloads/persistenceLayer.pdf

This Version: June 21, 2005

Copyright 1997-2005 Scott W. Ambler

Table Of Contents

1. GOOD THINGS TO KNOW ABOUT THIS PAPER 1

2. KINDS OF PERSISTENCE LAYERS 1

3. THE CLASS-TYPE ARCHITECTURE 3

4. REQUIREMENTS FOR A PERSISTENCE LAYER 5

5. THE DESIGN OF A PERSISTENCE LAYER 8

5.1 OVERVIEW OF THE DESIGN 8
5.1.1 The PersistentObject Class 9
5.1.2 The PersistentCriteria Class Hierarchy 10
5.1.3 The Cursor Class 12
5.1.4 The PersistentTransaction Class 13
5.1.5 The PersistenceBroker Class 14
5.1.6 The PersistenceMechanism Class Hierarchy 15
5.1.7 The Map Classes 16
5.1.8 The SqlStatement Class Hierarchy 18

6. IMPLEMENTING THE PERSISTENCE LAYER 19

6.1 BUY VERSUS BUILD 19
6.2 CONCURRENCY, OBJECTS, AND ROW LOCKING 19
6.3 DEVELOPMENT LANGUAGE ISSUES 20
6.4 A DEVELOPMENT SCHEDULE 21

7. DOING A DATA LOAD 21

7.1 TRADITIONAL DATA LOADING APPROACHES 21
7.2 ARCHITECTED DATA LOADING 22

8. SUPPORTING THE PERSISTENCE LAYER 23

9. SUMMARY 25

10. ABOUT THE AUTHOR 25

11. REFERENCES AND RECOMMENDED READING 26

Copyright 1997-2005 Scott W. Ambler

In this white paper I present an overview of the design of a robust persistence layer for object-oriented applications. I have implemented all or portions of this design in several languages, in other words, this design has been proven in practice.

1. Good Things to Know About This Paper

1. I assume that you have read my white paper entitled *Object/Relational Mapping 101* at www.agiledata.org/essays/mappingObjects.html.
2. Throughout this paper I will use the Unified Modeling Language (UML) version to represent my models.
3. Accessor methods, also known as getters and setters, are assumed for all attributes.
4. All attributes are private.
5. When I refer to an instance of class X, the implication is that I'm really referring to instances of class X or any of its subclasses. This concept is called the Liskov Substitution Principle.
6. I do not present code for the persistence layer (and I will not distribute it), nor do I go into language-specific issues in the design. I will however discuss implementation issues at the end of the paper.

2. Kinds of Persistence Layers

I would like to begin with a discussion of the common approaches to persistence that are currently in practice today. Figure 1 presents the most common, and least palatable, approach to persistence in which Structured Query Language (SQL) code is embedded in the source code of your classes. The advantage of this approach is that it allows you to write code very quickly and is a viable approach for small applications and/or prototypes. The disadvantage is that it directly couples your business classes with the schema of your relational database, implying that a simple change such as renaming a column or porting to another database results in a rework of your source code.

Hard-coded SQL in your business classes results in code that is difficult to maintain and extend.

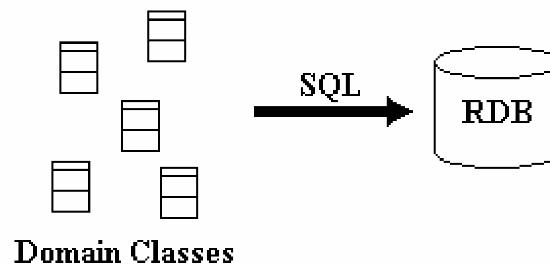

Figure 1. Hard-coding SQL in your domain/business classes.

Figure 2 presents a slightly better approach in which the SQL statements for your business classes are encapsulated in one or more "data classes." Once again, this approach is suitable for prototypes and small systems of less than 40 to 50 business classes but it still results in a recompilation (of your data classes) when simple changes to the database are made. Examples of this approach include developing stored procedures in the database to represent objects (replacing the data classes of Figure 2) and Enterprise JavaBean (EJB)'s entity bean strategy. The best thing that can be said about this approach is that you have at least encapsulated the source code that handles the hard-coded interactions in one place, the data classes.

Hardcoding SQL in separate data classes or stored procedures is only slightly better.

Copyright 1997-2005 Scott W. Ambler

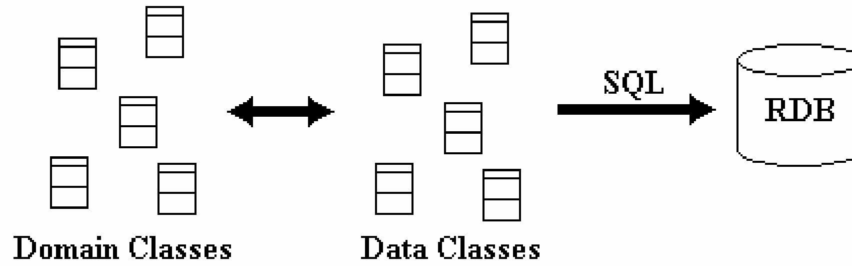

Figure 2. Creating data classes corresponding to domain/business classes.

Figure 3 presents the approach that will be taken in this paper, that of a robust persistence layer that maps objects to persistence mechanisms (in this case relational databases) in such a manner that simple changes to the relational schema do not affect your object-oriented code. The advantage of this approach is that your application programmers do not need to know a thing about the schema of the relational database, in fact, they don't even need to know that their objects are being stored in a relational database. This approach allows your organization to develop large-scale, mission critical applications. The disadvantage is that there is a performance impact to your applications, a minor one if you build the layer well, but there is still an impact.

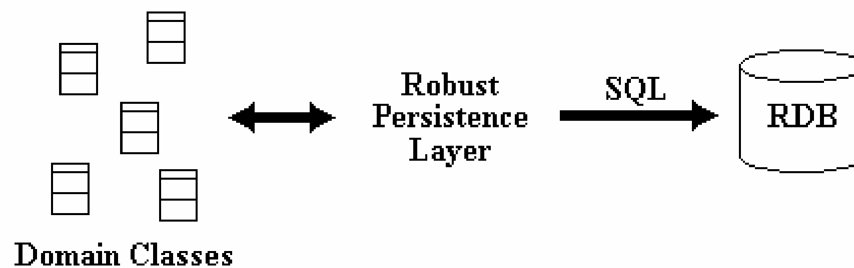

Figure 3. A robust persistence layer.

To understand our approach better, you must first understand the need for layering your application.

Copyright 1997-2005 Scott W. Ambler

3. The Class-Type Architecture

Figure 4 shows a class-type architecture (Ambler, 1998a; Ambler, 1998b, Ambler, 2004) that your programmers should follow when coding their applications. The class-type architecture is based on the Layer pattern (Buschmann, Meunier, Rohnert, Sommerlad, Stal, 1996), the basic idea that a class within a given layer may interact with other classes in that layer or with classes in an adjacent layer. By layering your source code in this manner you make it easier to maintain and to enhance because the coupling within your application is greatly reduced.

Layering your application code dramatically increases its robustness.

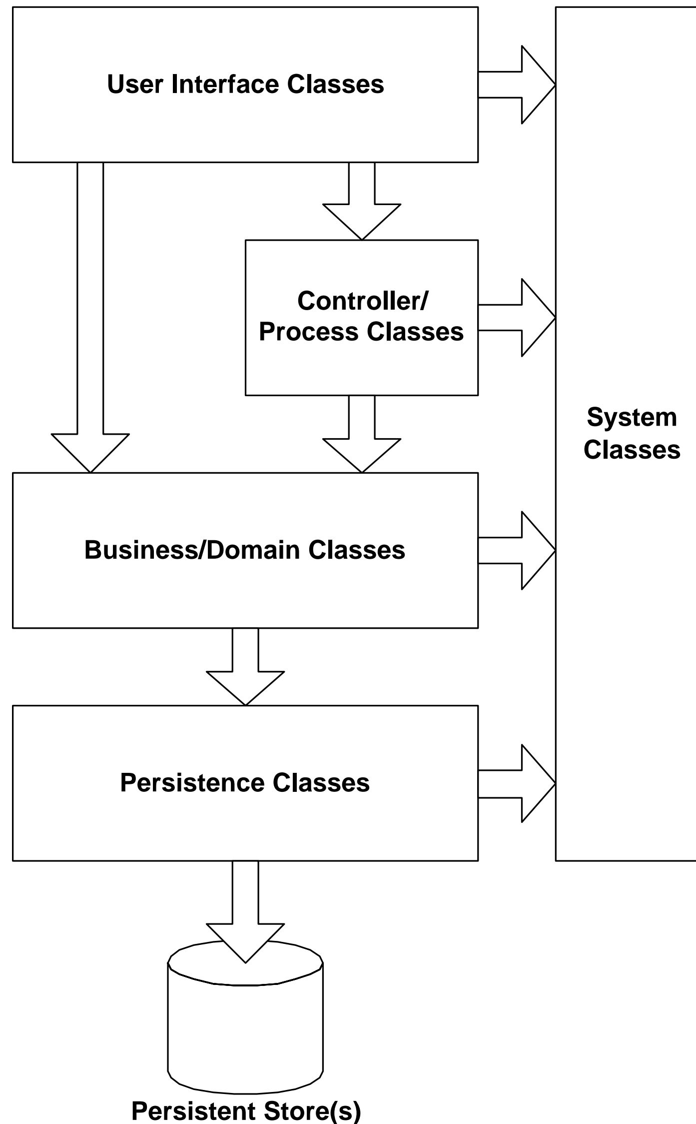

Figure 4. The class-type architecture.

Figure 4 indicates that users of your application interact directly with the user-interface layer of your application. The user-interface layer is generally made up of classes that implement screens and reports. User-interface classes are allowed to send messages to classes within the domain/business layer, the

Copyright 1997-2005 Scott W. Ambler

controller/process layer, and the system layer. The domain/business layer implements the domain/business classes of your application, for example the business layer for a telecommunications company would include classes such as **Customer** and **PhoneCall**. The controller/process layer, on the other hand, implements business logic that involves collaborating with several business/domain classes or even other controller/process classes such as the calculation of the charge of a phone call (which would interact with instances of **PhoneCall**, **Customer**, and **CallingPlan**). The system layer implements classes that provide access to operating system functionality such as printing and electronic mail. Domain/business classes are allowed to send messages to classes within the system layer and the persistence layer. The persistence layer encapsulates the behavior needed to store objects in persistence mechanisms such as object databases, files, and relational databases.

By conforming to this class-type architecture the robustness of your source code increases dramatically due to reduced coupling within your application. Figure 4 shows that for the user-interface layer to obtain information it must interact with objects in the domain/business layer, which in turn interact with the persistence layer to obtain the objects stored in your persistence mechanisms. This is an important feature of the class-type architecture – by not allowing the user interface of your application to directly access information stored in your persistence mechanism you effectively de-couple the user interface from the persistence schema. The implication is that you are now in a position to change the way that objects are stored, perhaps you want to reorganize the tables of a relational database or port from the persistence mechanism of one vendor to that of another, without having to rewrite your screens and reports.

Important heuristics:

1. **User-interface classes should not directly access your persistence mechanisms.** By encapsulating the business logic of your application in domain/business classes and controller/process classes, and not in your user interface, you are able to use that business logic in more than one place. For example, you could develop a screen that displays the total produced by an instance of the domain/business class *Invoice* (Ambler, 1998h) as well as a report that does the same. If the logic for calculating the total changes, perhaps complex discounting logic is added, then you only need to update the code contained within *Invoice* and both the screen and report will display the correct value. Had you implemented totaling logic in the user interface it would have been in both the screen and the report and you would need to modify the source code in two places, not just one.

2. **Domain/business classes should not directly access your persistence mechanisms.** Just like you do not want to allow user-interface classes to directly access information contained in your persistence mechanism, neither do you want to allow domain/business classes and controller/process to do so. We'll see in the next section that a good persistence layer protects your application code from persistence mechanism changes. If a database administrator decides to reorganize the schema of a persistence mechanism it does not make sense that you should have to rewrite your source code to reflect those changes.

3. **The class-type architecture is orthogonal to your hardware/network architecture.** An important thing to understand about the class-type architecture is that it is completely orthogonal to your hardware/network architecture. Table 1 shows how the various class types would be implemented on common hardware/network architectures. For example, we see that with the thin-client approach to client/server computing that user-interface and system classes are implemented on the client and that domain/business, persistence, and system classes are implemented on the server. Because system classes wrap access to network communication protocols you are guaranteed that some system classes will reside on each computer.

Copyright 1997-2005 Scott W. Ambler

Class Type	Stand Alone	Thin-Client	Fat-Client	n-Tier	Distributed Objects
User interface	Client	Client	Client	Client	Client
Controller/process	Client	Server	Client	Application server	Do not care
Domain/business	Client	Server	Client	Application server	Do not care
Persistence	Client	Server	Server	Database server	Do not care
System	Client	All machines	All machines	All machines	All machines

Table 1. Deployment strategies for class types for various hardware/network architectures.

4. Requirements For a Persistence Layer

I have always been a firm believer that the first thing you should do when developing software is define the requirements for it. The requirements presented here (Ambler, 1998d) reflect my experiences over the years building and using persistence layers. I first started working with the object paradigm in 1991, and since then I have developed systems in C++, Smalltalk, and Java for the financial, outsourcing, military, and telecommunications industries. Some of these projects were small, single-person efforts and some involved several hundred developers. Some were transaction-processing intensive whereas others dealt with very complex domains. The short story is that these requirements reflect my experiences on a diverse range of projects.

A persistence layer encapsulates the behavior needed to make objects persistent, in other words to read, write, and delete objects to/from permanent storage. A robust persistence layer should support:

1. **Several types of persistence mechanism.** A persistence mechanism is any technology that can be used to permanently store objects for later update, retrieval, and/or deletion. Possible persistence mechanisms include flat files, relational databases, object-relational databases, hierarchical databases, network databases, and objectbases. In this paper I will concentrate on the relational aspects of a persistence layer.

2. **Full encapsulation of the persistence mechanism(s).** Ideally you should only have to send the messages **save**, **delete**, and **retrieve** to an object to save it, delete it, or retrieve it respectively. That's it, the persistence layer takes care of the rest. Furthermore, except for well-justified exceptions, you shouldn't have to write any special persistence code other than that of the persistence layer itself.

3. **Multi-object actions.** Because it is common to retrieve several objects at once, perhaps for a report or as the result of a customized search, a robust persistence layer must be able to support the retrieval of many objects simultaneously. The same can be said of deleting objects from the persistence mechanism that meet specific criteria.

4. **Transactions.** Related to requirement #3 is the support for transactions, a collection of actions on several objects. A transaction could be made up of any combination of saving, retrieving, and/or deleting of objects. Transactions may be flat, an "all-or-nothing" approach where all the actions must either succeed or be rolled back (canceled), or they may be nested, an approach where a transaction is made up of other transactions which are committed and not rolled back if the large transaction fails. Transactions may also be short-lived, running in thousandths of a second, or long-lived, taking hours, days, weeks, or even months to complete.

5. **Extensibility.** You should be able to add new classes to your object applications and be able to change persistence mechanisms easily (you can count on at least upgrading your persistence mechanism over

Copyright 1997-2005 Scott W. Ambler

time, if not port to one from a different vendor). In other words your persistence layer must be flexible enough to allow your application programmers and persistence mechanism administrators to each do what they need to do.

6. **Object identifiers.** An object identifier (Ambler, 1998c), or OID for short, is an attribute, typically a number, that uniquely identifies an object. OIDs are the object-oriented equivalent of keys from relational theory, columns that uniquely identify a row within a table.

7. **Cursors.** A persistence layer that supports the ability to retrieve many objects with a single command should also support the ability to retrieve more than just objects. The issue is one of efficiency: Do you really want to allow users to retrieve every single person object stored in your persistence mechanism, perhaps millions, all at once? Of course not. An interesting concept from the relational world is that of a cursor. A cursor is a logical connection to the persistence mechanism from which you can retrieve objects using a controlled approach, usually several at a time. This is often more efficient than returning hundreds or even thousands of objects all at once because the user many not need all of the objects immediately (perhaps they are scrolling through a list).

8. **Proxies.** A complementary approach to cursors is that of a "proxy." A proxy is an object that represents another object but does not incur the same overhead as the object that it represents. A proxy contains enough information for both the computer and the user to identify it and no more. For example, a proxy for a person object would contain its OID so that the application can identify it and the first name, last name, and middle initial so that the user could recognize who the proxy object represents. Proxies are commonly used when the results of a query are to be displayed in a list, from which the user will select only one or two. When the user selects the proxy object from the list the real object is retrieved automatically from the persistence mechanism, an object which is much larger than the proxy. For example, the full person object may include an address and a picture of the person. By using proxies you don't need to bring all of this information across the network for every person in the list, only the information that the users actually want.

9. **Records.** The vast majority of reporting tools available in the industry today expect to take collections of database records as input, not collections of objects. If your organization is using such a tool for creating reports within an object-oriented application your persistence layer should support the ability to simply return records as the result of retrieval requests in order to avoid the overhead of converting the database records to objects and then back to records.

10. **Multiple architectures.** As organizations move from centralized mainframe architectures to 2-tier client/server architectures to n-tier architectures to distributed objects your persistence layer should be able to support these various approaches. The point to be made is that you must assume that at some point your persistence layer will need to exist in a range of potentially complex environments.

11. **Various database versions and/or vendors.** Upgrades happen, as do ports to other persistence mechanisms. A persistence layer should support the ability to easily change persistence mechanisms without affecting the applications that access them, therefore a wide variety of database versions and vendors should be supported by the persistence layer.

12. **Multiple connections.** Most organizations have more than one persistence mechanism, often from different vendors, that need to be accessed by a single object application. The implication is that a persistence layer should be able to support multiple, simultaneous connections to each applicable persistence mechanism. Even something as simple as copying an object from one persistence mechanism to another, perhaps from a centralized relational database to a local relational database, requires at least two simultaneous connections, one to each database.

13. **Native and non-native drivers.** There are several different strategies for accessing a relational database, and a good persistence layer will support the most common ones. Connection strategies

Copyright 1997-2005 Scott W. Ambler

include using Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), and native drivers supplied by the database vendor and/or a third party vendor.

14. **Structured query language (SQL) queries.** Writing SQL queries in your object-oriented code is a flagrant violation of encapsulation – you've coupled your application directly to the database schema. However, for performance reasons you sometimes need to do so. Hard-coded SQL in your code should be the exception, not the norm, an exception that should be well-justified before being allowed to occur. Anyway, your persistence layer will need to support the ability to directly submit SQL code to a relational database.

Persistence layers should allow application developers to concentrate on what they do best, develop applications, without having to worry about how their objects will be stored. Furthermore, persistence layers should also allow database administrators (DBAs) to do what they do best, administer databases, without having to worry about accidentally introducing bugs into existing applications. With a well-built persistence layer DBAs should be able to move tables, rename tables, rename columns, and reorganize tables without affecting the applications that access them. Nirvana? You bet. My experience is that it is possible to build persistence layers that fulfill these requirements, in fact the design is presented below.

Copyright 1997-2005 Scott W. Ambler

5. The Design of a Persistence Layer

In this section I will present the design of a robust persistence layer. In a later section I will discuss the implementation issues associated with this design.

5.1 Overview of the Design

Figure 5 presents a high-level design (Ambler, 1998b) of a robust persistence layer and Table 2 describes each class in the figure. An interesting feature of the design is that an application programmer only needs to know about the following classes to make their objects persistent: **PersistentObject**, the **PersistentCriteria** class hierarchy, **PersistentTransaction**, and **Cursor**. The other classes are not directly accessed by application development code but will still need to be developed and maintained to support the "public" classes.

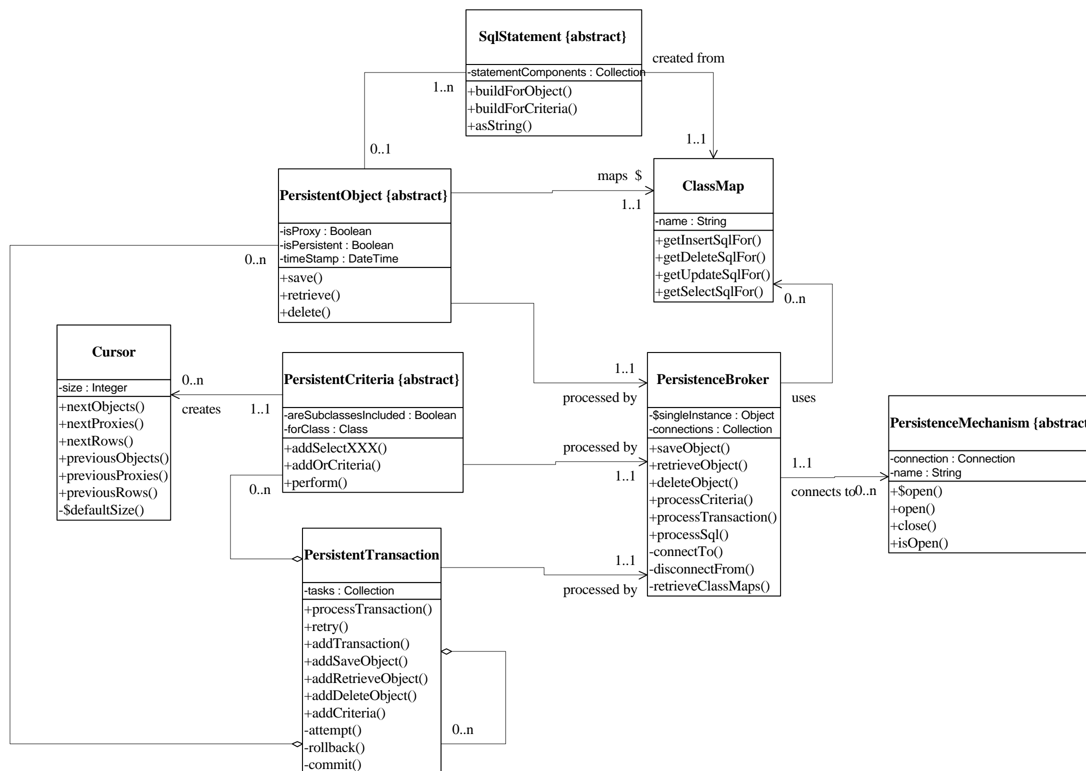

Figure 5. Overview of the design for a persistence layer.

Class	Description
ClassMap	A collection of classes that encapsulate the behavior needed to map classes to relational tables.
Cursor	This class encapsulates the concept of a database cursor.
PersistenceBroker	Maintains connections to persistence mechanisms, such as relational databases and flat files, and handles the communication between the object application and the persistence mechanisms.
PersistentCriteria	This class hierarchy encapsulates the behavior needed to retrieve, update, or delete collections of objects based on defined criteria.
PersistenceMechanism	A class hierarchy that encapsulates the access to flat files, relational databases, and object-relational databases. For relational databases this hierarchy wraps complex class libraries, such as Microsoft's ODBC (open database connectivity) or Java's JDBC (Java database connectivity), protecting your organization from changes to the class libraries.
PersistentObject	This class encapsulates the behavior needed to make single instances persistent and is the class that business/domain classes inherit from to become persistent.
PersistentTransaction	This class encapsulates the behavior needed to support transactions, both flat and nested, in the persistence mechanisms.
SqlStatement	This class hierarchy knows how to build insert, update, delete, and select SQL (structured query language) statements based on information encapsulated by **ClassMap** objects.

Table 2. The classes and hierarchies of the persistence layer.

The classes represented in Figure 5 each represent cohesive concepts, in other words each class does one thing and one thing well. This is a fundamental of good design. **PersistentObject** encapsulates the behavior needed to make a single object persistent whereas the **PersistentCriteria** class hierarchy encapsulates the behaviors needed to work with collections of persistent objects. Furthermore, I'd like to point out that the design presented here represents my experiences building persistence layers in Java, C++, and Smalltalk for several problem domains within several different industries. This design works and is proven in practice by a wide range of applications.

5.1.1 The PersistentObject Class

Figure 6 shows the design of two classes, **PersistentObject** and **OID**. **PersistentObject** encapsulates the behavior needed to make a single object persistent and is the class from which all classes in your problem/business domain inherit from. For example, the business class **Customer** will either directly or indirectly inherit from **PersistentObject**. The **OID** class encapsulates the behavior needed for object IDs, called persistent IDs in the CORBA (Common Object Request Broker) community, using the HIGH/LOW approach for ensuring unique identifiers. Details of the HIGH/LOW OID are presented at www.agiledata.org.

Copyright 1997-2005 Scott W. Ambler

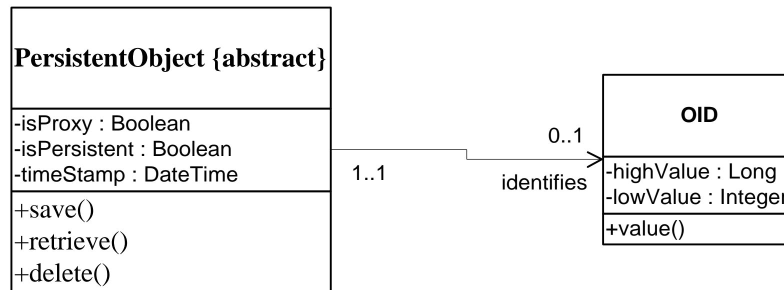

Figure 6. The design of PersistentObject and OID.

As you can see, **PersistentObject** is fairly simple. It has three attributes, **isProxy**, **isPersistent**, and **timeStamp** which respectively indicate whether or not an object is a proxy, if it was retrieved from a persistence mechanism, and the **timeStamp** assigned by the persistence mechanism for when it was last accessed by your application. Proxy objects include only the minimal information needed for the system and the user to identify the object, therefore they reduce network traffic as they are smaller than the full objects. When the "real" object is needed the proxy is sent the **retrieve()** message which refreshes all of the object's attributes. Proxies are used when the user is interested in a small subset of the objects that would be the result of a retrieval, often the case for a search screen or simple list of objects. The attribute **isPersistent** is important because an object needs to know if it already exists in the persistence mechanism or if it was newly created, information that is used to determine if an insert or update SQL statement needs to be generated when saving the object. The **timeStamp** attribute is used to support optimistic locking in the persistence mechanism. When the object is read into memory its **timeStamp** is updated in the persistence mechanism. When the object is subsequently written back the **timeStamp** is first read in and compared with the initial value – if the value of **timeStamp** has changed then another user has worked with the object and there is effectively a collision which needs to be rectified (typically via the display of a message to the user).

PersistentObject implements three methods – **save()**, **delete()**, **and retrieve()** – messages which are sent to objects to make them persistent. The implication is that application programmers don't need to have any knowledge of the persistence strategy to make objects persistent, instead they merely send objects messages and they do the right thing. This is what encapsulation is all about.

PersistentObject potentially maintains a relationship to an instance of **OID**, which is done whenever object IDs are used for the unique keys for objects in the persistence mechanism. This is optional because you don't always have the choice to use object IDs for keys, very often you are forced to map objects to a legacy schema. The need to map to legacy schemas is an unfortunate reality in the object-oriented development world, something that we'll discuss later in this white paper we look at how the map classes are implemented. Anyway, you can easily have **PersistentObject** automatically assign object IDs to your objects when they are created if you have control over your persistence schema.

5.1.2 The PersistentCriteria Class Hierarchy

Although **PersistentObject** encapsulates the behavior needed to make single objects persistent, it is not enough because we also need to work with collections of persistent objects. This is where the **PersistentCriteria** class hierarchy of Figure 7 comes in – it supports the behavior needed to save, retrieve, and delete several objects at once.

Copyright 1997-2005 Scott W. Ambler

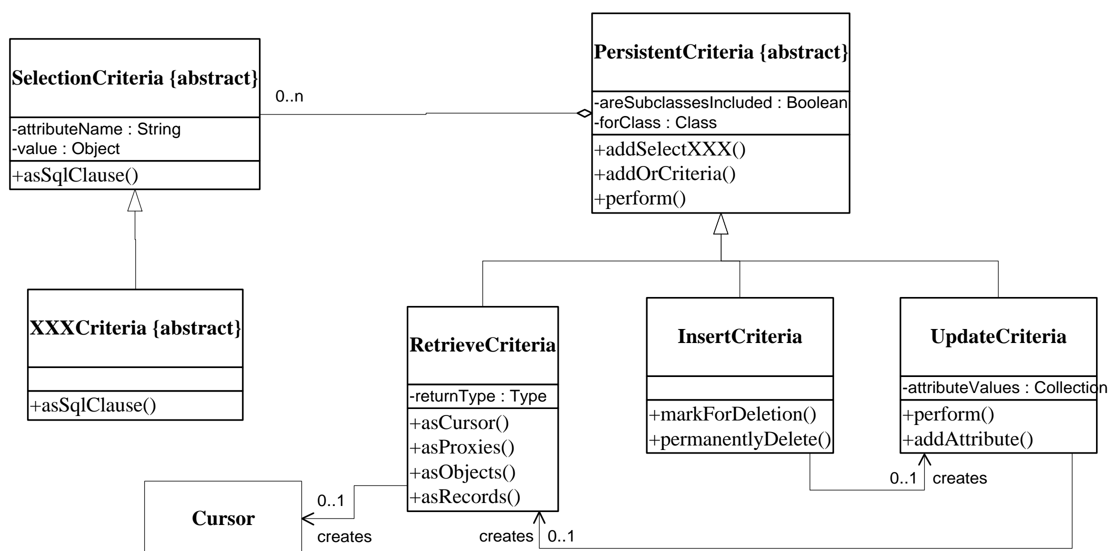

Figure 7. The PersistentCriteria class hierarchy.

PersistentCriteria is an abstract class, one that captures behavior common to its subclasses but one that is not directly instantiated, which allows you to define selection criteria that limits the scope to a small subset of objects. The **addSelectXXX()** method of **PersistentCriteria** represents a collection of methods that take two parameters, an attribute of a class and a value, and create corresponding instances of subclasses of **SelectionCriteria**. The **SelectionCriteria** class hierarchy encapsulates the behavior needed to compare a single attribute to a given value. There is one subclass for each basic type of comparison (equal to, greater than, less than, less than or equal to, and greater than or equal to). For example, the method **addSelectGreaterThan**() method creates an instance of **GreaterThanCriteria**, and **addSelectEqualTo()** creates an instance of **EqualToCriteria**.

The **forClass** attribute of **PersistentCriteria** indicates the type of objects being dealt with, perhaps **Employee** or **Invoice** objects, and the **isSubclassesIncluded** attribute indicates whether or not the criteria also applies to subclasses of **forClass**, effectively supporting inheritance polymorphism. The combination of these two attributes and the **addSelectXXX()** methods are what makes it possible to define that you want to work with instances of the **Person** class and it subclasses where their first names begin with the letter 'J' (through wild card support) that were born between June 14th, 1966 and August 14th 1967.

The class **RetrieveCriteria** supports the retrieval of zero or more objects, proxy objects, rows, or a cursor because we want to be able to retrieve more than just objects: Proxies are needed to reduce network traffic, rows are needed because many reporting class libraries want collections of rows (not real objects) as parameters, and cursors allow you to deal with small subsets of the retrieval result set at a time increasing the responsiveness of your application. The **Cursor** class will be discussed later.

DeleteCriteria supports the deletion of several objects at once. This robust class supports both marking objects as deleted, my preferred approach, and actually deleting of them (perhaps to clean up the database and/or for archiving). To mark objects as deleted the instance of **DeleteCriteria** creates an instance of **UpdateCriteria** and simply updates a **deletionDateTime** or **isDeleted** column within the appropriate tables.

The class **UpdateCriteria** is used to update one or more attributes within a collection of classes simultaneously. The **perform()** method basically creates an instance of **RetrieveCriteria** to obtain the objects, loops through them to assign the new values to the attributes, and then sends the **save()** message to

Copyright 1997-2005 Scott W. Ambler

each object to write it back to the persistence mechanism. You need to retrieve the objects so that you can use the appropriate setter methods to update the attributes – the setter methods will ensure that the applicable business rules are followed when the new values are set. Remember, objects encapsulate business rules which are often not reflected in the database, therefore you cannot simply generate a single SQL statement to update all objects at once.

The typical life cycle of a persistent criteria object is to define zero or more selection criteria for it and then to have the object run itself (it submits itself to the single instance of **PersistenceBroker**) via the **perform()** method. Instances of **SelectionCriteria** are related to one another within a single instance of **PersistentCriteria** via the use of "AND logic." To support OR logic the **orCriteria()** method takes an instance of **PersistentCriteria** as a parameter and effectively concatenates the two criteria together. As you would guess, this makes it possible to generate very complex criteria objects.

The advantage of this class hierarchy is that it allows application programmers to retrieve, delete, and update collections of objects stored within a persistence mechanism without having any knowledge of the actual schema. Remember, the **SelectionCriteria** class deals with the attributes of objects, not with columns of tables. This allows application programmers to build search screens, lists, and reports that aren't coupled to the database schema, and to archive information within a persistence mechanism without direct knowledge of its design. Once again, our persistence layer supports full encapsulation of the persistence mechanism's schema.

5.1.3 The Cursor Class

Figure 8 shows the design of the **Cursor** class which encapsulates the basic functionality of a database cursor. Cursors allow you to retrieve subsets of information from your persistence mechanism at a single time. This is important because a single retrieve, supported by the **RetrieveCriteria** class described last month, may result in hundreds or thousands of objects coming across the network – by using a cursor you can retrieve this result set in small portions one at a time. **Cursor** objects allow you to traverse forward and backward in the result set of a retrieval (most databases support forward traversal but may not support reverse traversal due to server buffering issues), making it easy to support users scrolling through lists of objects. The **Cursor** class also supports the ability to work with rows (records) from the database, proxy objects, and full-fledged objects.

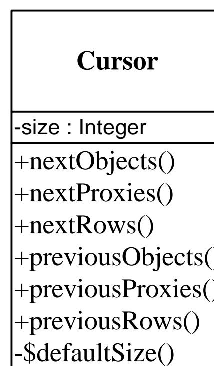

Figure 8. The Cursor class.

Cursor has an instance attribute **size**, whose value is typically between one and fifty, which indicates the maximum number of rows, objects, or proxies that will be brought back at a single time. As you would expect, the class/static method **defaultSize()** returns the default cursor size, which I normally set at one. Note how a getter method for the default size is used, not a constant (static final for the Java programmers

out there). By using a getter method to obtain the constant value I leave open the opportunity for calculating the value, instead of just hardcoding it as a constant. I argue that the principle of information hiding pertains to constants as well as variables, therefore I use getter methods for constants to make my code more robust.

5.1.4 The PersistentTransaction Class

The fourth and final class that your application programmers will directly deal with – the others were **PersistentObject**, **PersistentCriteria**, and **Cursor** – is **PersistentTransaction**, shown in Figure 9. **PersistentTransaction** instances are made up of tasks to occur to single objects, such as saving, deleting, and retrieving them, as well as instances of **PersistentCriteria** and other **PersistentTransaction** objects.

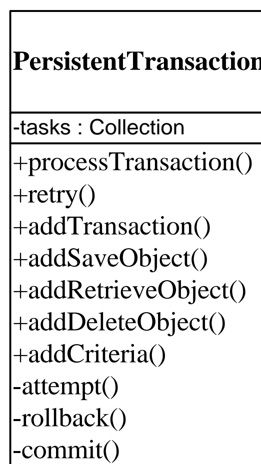

Figure 9. The PersistentTransaction class.

The typical life cycle of a transaction is to create it, add a series of tasks, send it the **processTransaction()** message, and then either commit the transaction, rollback the transaction, or retry the transaction. You would commit the transaction, make the tasks of the transaction permanent, only if the **processTransaction()** method indicated that the transaction was successful. Otherwise, you would either rollback the transaction, basically give up trying the transaction, or retry the transaction if it's possible that locks in your persistence mechanism have been removed (making it possible to succefully run the transaction). The ability to commit and rollback transactions is important – because transactions are atomic, either the succeed or they fail – you must be able to either completely back out of the transaction by rolling it back or completely finish the transaction by committing it.

Tasks are processed in the order that they are added to an instance of **PersistentTransaction**. If a single task fails, perhaps it is not possible to delete an indicated object, then processing stops at that task and the **processTransaction()** method returns with a failure indication.

When a **PersistentTransaction** instance is added to another transaction, via invoking the **addTransaction()** method, it is considered to be nested within the parent transaction. Child transactions can be successful, be committed, even when the parent transaction fails. When a nested transaction is attempted, if it is successful it is automatically committed before the next task in the list is attempted, otherwise if it fails the parent transaction stops with a failure indication.

Copyright 1997-2005 Scott W. Ambler

An advanced version of this class would allow for non-persistence mechanism tasks to be included in a transaction. For example, perhaps it's important to run a transaction only on days where the moon is full, therefore one of your transaction steps would be to send the message **isFull()** to an instance of the **Moon** class, if **isFull()** returns true then the transaction continues, otherwise it fails.

5.1.5 The PersistenceBroker Class

In many ways the **PersistenceBroker** class, show in Figure 10, is the key to the persistence layer. This class follows the Singleton design pattern in that there is only one instance of it in the object space of the application. During run time **PersistenceBroker** maintains connections to persistence mechanisms (databases, files, …) and manages interactions with them. **PersistenceBroker** effectively acts as a go between for the classes **PersistentObject**, **PersistentCriteria**, and **Transaction** as it is where instances of these classes submit themselves to be processed. **PersistenceBroker** interacts with the **SqlStatement** class hierarchy, map classes, and **PersistenceMechanism** class hierarchy.

PersistenceBroker
-$singleInstance : Object -connections : Collection
+saveObject() +retrieveObject() +deleteObject() +processCriteria() +processTransaction() +processSql() -connectTo() -disconnectFrom() -retrieveClassMaps()

Figure 10. The PersistenceBroker class.

When you start your application one of the initiation tasks is to have **PersistenceBroker** read in the information needed to create instances of the map classes (**ClassMap**, **AttributeMap**, …) from your persistence mechanism. **PersistenceBroker** then buffers the map classes in memory so they can be used to map objects into the persistence mechanism.

An important feature of **PersistenceBroker** is the **processSql()** method, which you can use to submit hardcoded SQL (structured query language) statements to the persistence. This is a critical feature because it allows you to embed SQL in your application code – when performance is of critical importance you may decide to override the **save()**, **delete()**, and/or **retrieve()** methods inherited from **PersistentObject** and submit SQL directly to your persistence mechanism. Although this always sounds like a good idea at the time, it is often a futile effort for two reasons: first, the resulting increase in coupling between your application and the persistence schema reduces the maintainability and extensibility of your application; second, when you actually profile your application to discover where the processing is taking place it is often in your persistence mechanism, not in your persistence layer. The short story is that to increase the performance of your application your time is better spent tweaking the design of your persistence schema, not your application code.

Copyright 1997-2005 Scott W. Ambler

5.1.6 The PersistenceMechanism Class Hierarchy

The **PersistenceMechanism** class hierarchy, shown in Figure 11, encapsulates the behaviors of the various kinds of persistence mechanisms. Although support for object-relational databases and files is shown here, we're concentrating on mapping objects to relational databases. Flat files in general provide less functionality than relational databases, basically the sequential reading and writing of data, whereas object-relational databases provide more.

The class method (static method in Java and C++) **open( )** is effectively a constructor method that takes as a parameter the name of a persistence mechanism to connect to, answering back the corresponding instance of **PersistenceMechanism**.

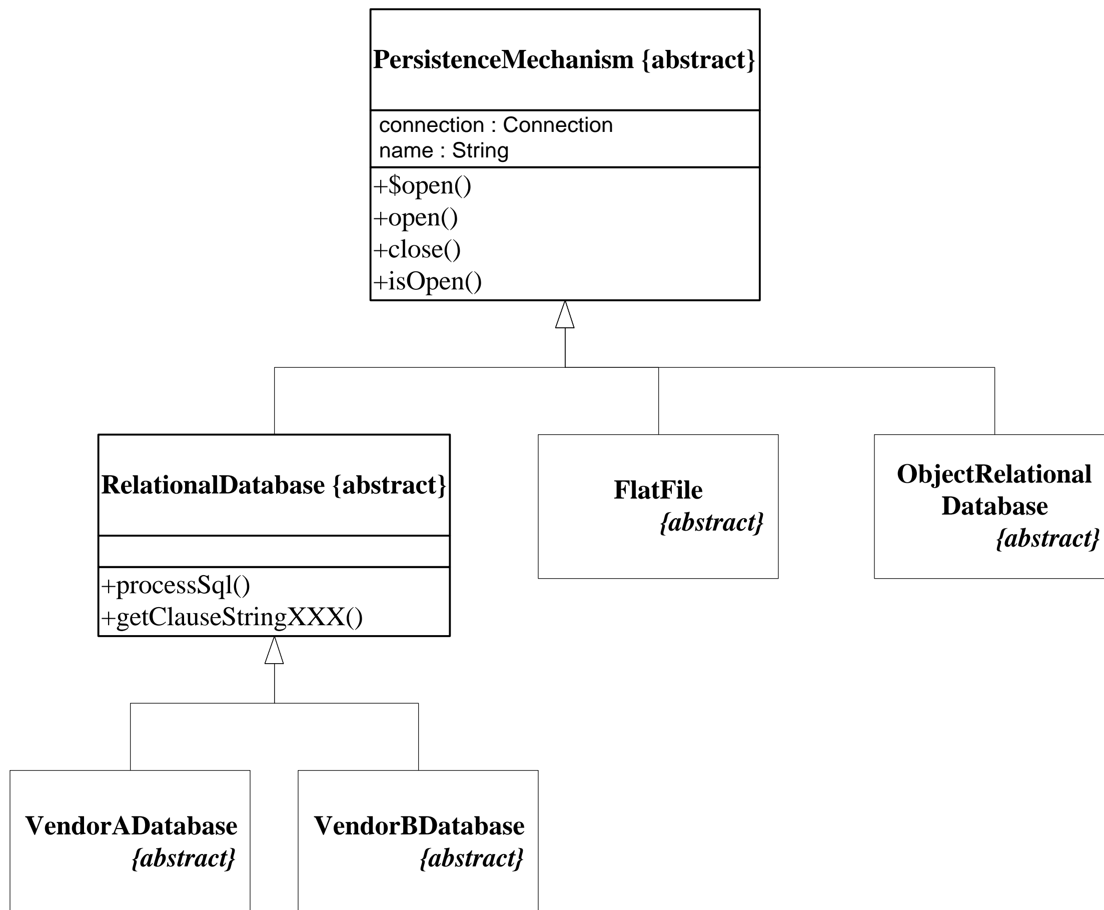

Figure 11. **The PersistenceMechanism class hierarchy.**

The **getClauseStringXXX()** of **RelationalDatabase** represents a series of getter methods that return strings representing a portion of a SQL statement clause (this information is used by the **SqlStatement** class hierarchy). Examples of XXX include: **Delete**, **Select**, **Insert**, **OrderBy**, **Where**, **And**, **Or**, **Clause**, **EqualTo**, and **Between**. Often there will be two versions of each method, for example **And** really needs an **getClauseStringAndBegin()** method that returns the string 'AND(' and **getClauseStringAndEnd()** which returns the string ')' in order to build a complete AND clause within an SQL statement. These methods are invoked by instances of the **SqlStatement** class hierarchy so that they may take advantage of the unique features of each kind of relational database.

Copyright 1997-2005 Scott W. Ambler

RelationalDatabase supports the ANSI standard SQL clauses, whereas its subclasses will override the appropriate methods to support their own unique extensions to ANSI SQL. This class, and its subclasses, wrap complex class libraries such as Microsoft's ODBC (open database connectivity) or Java's JDBC (Java database connectivity), protecting your organization from changes to the class libraries. The method **processSQL()** takes as input a string representing an SQL statement and returns either a result set of zero or more rows or an error indicating a problem. This method is invoked only by **PersistenceBroker**, which maintains connections to your persistence mechanisms, and not by your application code which knows nothing about this class hierarchy (nor should it).

5.1.7 The Map Classes

Figure 12 presents the class diagram for the **ClassMap** component, a collection of classes that encapsulate the behavior needed to map objects to persistence mechanisms. The design is geared toward mapping objects to relational databases, although you can easily enhance it to support other persistence mechanisms such as flat files and object-relational databases.

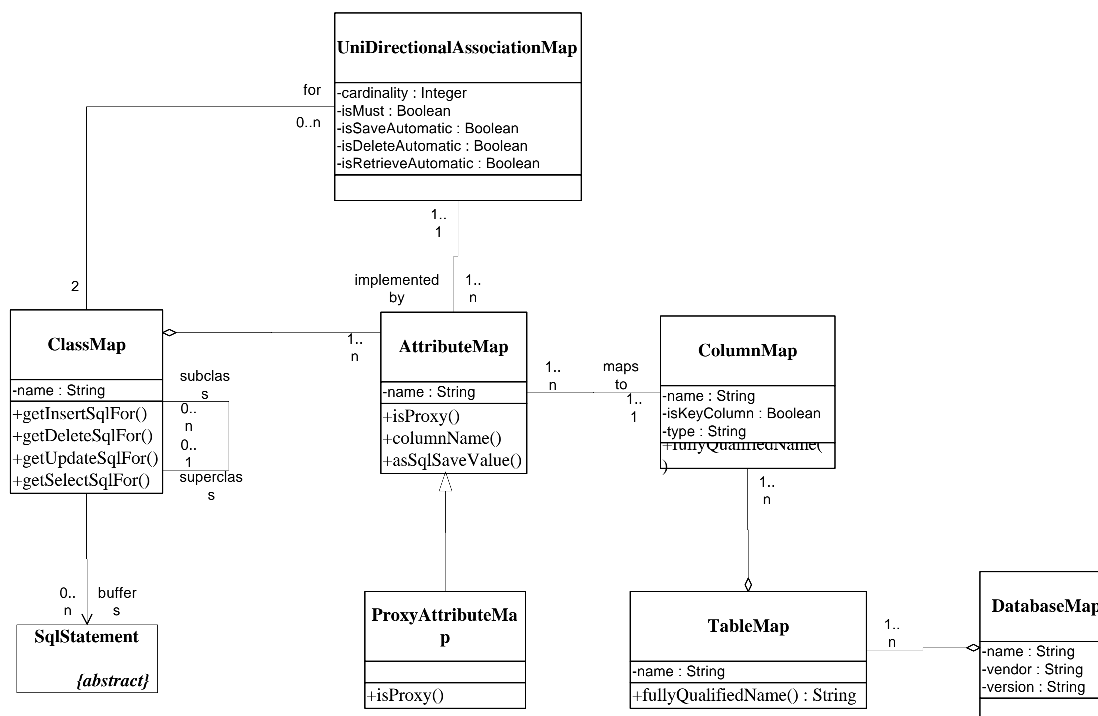

Figure 12. The ClassMap component.

Let's start at the **ClassMap** class, instances of which encapsulate the behavior needed to map instances of a given class to a relational database. If instances of the **Customer** class are persistent then there would be **ClassMap** object which maps **Customer** objects into the database. If instances of a class are not persistent, for examples instances of the class **RadioButton** (a user interface widget), then there will not be an instance of **ClassMap** for that class.

ClassMap objects maintain a collection of **AttributeMap** objects which may map an attribute to a single column in a relational table. **AttributeMap** objects map simple attributes such as strings and numbers that are stored in your database, or are used to represent collections to support instances of the **UniDirectionalAssociationMap** class (more on this in a minute). **AttributeMap** objects know what **ColumnMap** objects they are associated to, which in turn know their **TableMap** and **DatabaseMap**

Copyright 1997-2005 Scott W. Ambler

objects. Instances of these four classes are used to map an attribute of an object to a table column within a relational database.

A **ProxyAttributeMap** object is used to map a proxy attribute, which is an attribute that is needed to build the proxy version of an object. Proxy objects have just enough information to identify the real object that it represents, forgoing the values of attributes which require significant resources such as network bandwidth and memory. The **ProxyAttributeMap** class is needed to support the ability for **PersistentCriteria** objects and **Cursor** objects to automatically retrieve proxies from the database.

The class **UniDirectionalAssociationMap** encapsulates the behavior for maintaining a relationship between two classes. When a relationship is bi-directional, for example a **Student** object needs to know the courses that it takes and a **Course** object needs to know the students taking it, then you will need to maintain a **UniDirectionalAssociationMap** for each direction of the relationship. You could attempt to develop a **BiDirectionalAssociationMap** class if you wish, but when you consider the complexities of doing so you'll recognize that using two instances of **UniDirectionalAssociationMap** is much easier. The map maintains a relationship between two classes, and includes knowledge of whether or not the second class should be saved, deleted, or retrieved automatically when the first class is, effectively simulating triggers in your OO application (removing the need to maintain them in your database if you wish to do so).

The implemented by association between **UniDirectionalAssociationMap** and **AttributeMap** reveals the most interesting portion of this component – sometimes **AttributeMap** objects are used to represent a collection attribute to maintain a one-to-many association. For example, because a student takes one or more courses there is a one-to-many association from the **Student** class to the **Course** class. To maintain this association in your object application the **Student** class would have an instance attribute called **courses** which would be a collection of **Course** objects. Assuming the **isRetrieveAutomatic** attribute is set to true, then when a **Student** object is retrieved all of the courses that the student takes would be retrieved and references to them would be inserted into the collection automatically. Similar to defining triggers in relational databases, you want to put a lot of thought into the triggers that you define using the **isSaveAutomatic**, **isRetrieveAutomatic**, and **isDeleteAutomatic** attributes of **UniDirectionalAssociationMap**.

Why do you need these mapping classes? Simple, they are the key to encapsulating your persistence mechanism schema from your object schema (and vice versa). If your persistence mechanism schema changes, perhaps a table is renamed or reorganized, then the only change you need to make is to update the map objects, which as we'll see later are stored in your database. Similarly, if you refactor your application classes then the persistence mechanism schema does not need to change, only the map objects. Naturally, if new features are added requiring new attributes and columns, then both schemas would change, along with the maps, to reflect these changes.

For performance reasons instances of **ClassMap** maintain a collection of **SqlStatement** objects, buffering them to take advantage of common portions of each statement. For similar reasons, although I don't show it, **ClassMap** should also maintain a collection of **Database Map** objects that **SqlStatement** objects use to determine the proper subclass of **RelationalDatabase**, for example **Oracle8**, to obtain the specific string portions to build themselves. Without this relationship the **SqlStatement** objects need to traverse the relationships between the map classes to get to the right subclass of **RelationalDatabase**.

There are two interesting lessons to be learned from the class diagram in Figure 12. First, is the cardinality of "2" used on the association between **ClassMap** and **UniDirectionalAssociationMap** – I rarely indicate a maximum cardinality on an association, but this is one of the few times that a maximum is guaranteed to hold (there will only ever be two classes involved in a uni-directional association). The modeling of maximums, or minimums for that matter, is generally a bad idea because they will often change, therefore you don't want to develop a design that is dependent on the maximum. Second, recursive relationships are one of the few times that I use roles in an association – many people find recursive relationships confusing,

Copyright 1997-2005 Scott W. Ambler

such as the one that **ClassMap** has with itself, so you want to provide extra information to aid them in their understanding.

5.1.8 The SqlStatement Class Hierarchy

Figure 13 presents the **SqlStatement** class hierarchy which encapsulates the ability to create SELECT, INSERT, UPDATE, and DELETE structured query language (SQL) statements. As you would expect, each subclass knows how to build itself for a given object or instance of **PersistentCriteria**. For example, **SelectSqlStatement** objects will be created to retrieve a single **Customer** object, via invoking the **retrieve()** method on the object, or by creating an instance of the class **RetrieveCriteria**, a subclass of **PersistentCriteria**, and invoking the **perform()** method on it.

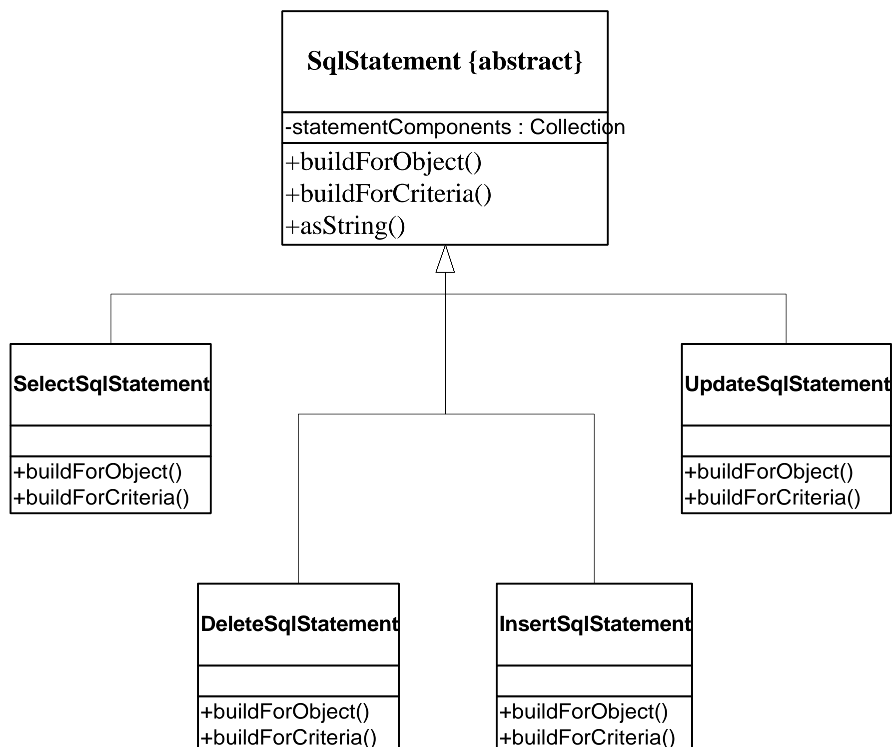

Figure 13. The SqlStatement class hierarchy.

As we saw earlier the **RelationalDatabase** class hierarchy encapsulates the specific flavor of SQL supported by each database vendor/version (although SQL is a standard, every vendor supports its own unique extensions that we want to automatically use). Instances of **SqlStatement** collaborate with instances of **ClassMap** to determine the subclass of **RelationalDatabase** from which to retrieve the portions of SQL clauses to build itself.

The attribute **statementComponents** is a collection of strings that can be reused for the single objects of a given class. For example, the attribute list of an INSERT statement does not change between instances of the same class, nor does the INTO clause.

Copyright 1997-2005 Scott W. Ambler

6. Implementing The Persistence Layer

There are several issues that you need to be aware of with persistence layers if you wish to be successful. These issues are:

? Buying versus building the persistence layer
? Concurrency, objects, and row locking
? Development language issues
? A potential development schedule

6.1 Buy Versus Build

Although this white paper is aimed at people who are building a persistence layer, the fact is that building and maintaining a persistence layer is a complex task. My advice is that you shouldn't start the development of a persistence layer it if you can't finish through. This includes the maintenance and support of the persistence layer once it is in place.

If you decide that you either can't or don't want to build a persistence layer then you should consider purchasing once. In my third book, *Process Patterns* (Ambler, 1998b), I go into detail about the concept of a feasibility study, which looks at the economic, technical, and operational feasibility of something. The basic idea is that your persistence layer should pay for itself, should be possible to build/buy, and should be possible to be supported and maintained over time (as indicated previously).

A feasibility study should look at the economic, technical, and operational feasibility of building/buying a persistence layer.

The good news is that there are a lot of good persistence products available on the market, and I have provided links to some of them at http://www.ambysoft.com/persistenceLayer.html to provide an initial basis for your search. Also, I have started, at least at a high level, a list of requirements for you in this document for your persistence layer. The first thing that you need to do is flesh them out and then prioritize them for your specific situation.

6.2 Concurrency, Objects, and Row Locking

For the sake of this white paper concurrency deals with the issues involved with allowing multiple people simultaneous access to the same record in your relational database. Because it is possible, if you allow it, for several users to access the same database records, effectively the same objects, you need to determine a control strategy for allowing this. The control mechanism used by relational databases is locking, and in particular row locking. There are two main approaches to row locking: pessimistic and optimistic.

1. **Pessimistic locking**. An approach to concurrency in which an item is locked in the persistence mechanism for the entire time that it is in memory. For example, when a customer object is edited a lock is placed on the object in the persistence mechanism, the object is brought into memory and edited, and then eventually the object is written back to the persistence mechanism and the object is unlocked. This approach guarantees that an item won't be updated in the persistence mechanism while the item is in memory, but at the same time is disallows others to work with it while someone else does. Pessimistic locking is ideal for batch jobs that need to ensure consistency in the data that they write.

2. **Optimistic locking**. An approach to concurrency in which an item is locked in the persistence mechanism only for the time that it is accessed in the persistence mechanism. For example, if a customer object is edited a lock is placed on it in the persistence mechanism for the time that it takes to read it in memory and then it is immediately removed. The object is edited and then when it needs to be saved it is locked again, written out, then unlocked. This approach allows many people to work with an object simultaneously, but also presents the opportunity for people to overwrite the work of others. Optimistic locking is best for online processing.

Copyright 1997-2005 Scott W. Ambler

Yes, with optimistic locking you have an overhead of determining whether or not the record has been updated by someone else when you go to save it. This can be accomplished via the use of a common timestamp field in all tables: When you read a record you read in the timestamp. When you go to write the record you compare the timestamp in memory to the one in the database, if they are the same then you update the record (including the timestamp to the current time). If they are different the someone else has updated the record and you can't overwrite it (therefore displaying a message to the user).

6.3 Development Language Issues

The design as presented in this paper requires something called reflection, the ability to work with objects dynamically at run time. Reflection is needed to dynamically determine the signatures of, based on the meta data contained in the map classes, getter and setter methods and then to invoke them appropriately. Reflection is built into languages such as Smalltalk and Java (at least for JDK 1.1+) but not (yet) in C++. The result is that in C++ you need to code around the lack of reflection, typically by moving collections of data between the business/domain layer and the persistence layer in a structured/named approach. As you would expect, this increases the coupling between your object schema and your data schema, although still provides you with some protection.

Copyright 1997-2005 Scott W. Ambler

6.4 A Development Schedule

If you intend to build a persistence layer, here is one potential schedule that you may choose to follow:

Milestone	Tasks to Perform
1. Implement basic CRUD behavior.	? Implement **PersistentObject**. ? Implement connection management in **PersistenceBroker**. ? Implement map classes (at least the basics) with the meta data being read from tables where the data is input manually. ? Implement basics of the **SqlStatement** hierarchy for a single object. ? Implement the **PersistenceMechanism** hierarchy for the database(s) that need to be supported within your organization.
2. Implement support for Associations.	? Implement the **UniDirectionalAssociationMap** class. ? The **SqlStatement** hierarchy will need to be updated to reflect the additional complexity of building SQL code to support associations.
3. Implement support for PersistentCriteria.	? Implement the **PersistentCriteria** hierarchy, typically starting with **RetrieveCriteria** to support search screens and reports. ? Update **PersistenceBroker** to process **PersistentCriteria** objects.
4. Implement support for cursors, proxies, and records.	? Add **ProxyAttributeMap**. ? Add **Cursor** class. ? Add **Record** class (if your language doesn't already support it). ? Add **Proxy** class (if your language doesn't already support it). ? Modify **PersistenceBroker** to hand back objects, rows, proxies, or records when processing **PersistentCriteria** objects.
5. Implement an administration application.	? See section 8.
6. Implement transactions.	? Implement the **Transaction** class. ? Modify **PersistenceBroker**.

I always suggest starting simple by supporting a single database, and then if needed support multiple databases simultaneously.

Steps 2 through 6 could be done in any order depending on your priorities.

7. Doing a Data Load

In this section I will discuss the issues involved with loading data into your object-oriented application. Data loads are a reality of system development: you need to convert a legacy database to a new version; you need to load testing/development objects from an external data source; or you need to perform regular loads, potentially in real time, of data from non-OO and/or external systems. I begin by reviewing the traditional loading techniques, and then present one that is sensible for OO applications.

7.1 Traditional Data Loading Approaches

The traditional approach to data loading, shown in Figure 14, is to write a program to read data in from the source database, cleanse it, then write it out to the target database. Cleansing may range from simple normalization of data, to single field cleansing such as converting two-digit years to four-digit years, to multi-field cleansing in which the value in one field implies the purpose of another field (yes, this would be

Copyright 1997-2005 Scott W. Ambler

considered incredibly bad design within the source data, but it is the norm in many legacy databases). Referential integrity, the assurance that all references within a record to other records do in fact refer to existing records, is also coded in the data loading program.

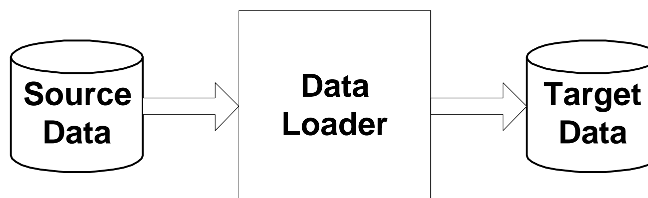

Figure 14. The traditional approach to loading data.

There are several problems with this approach. First and foremost, the target data is no longer encapsulated – if the schema of your persistence mechanism changes then you will need to change your data loader code. Granted, this can be alleviated by data loading tools that operate on meta data (they effectively have a persistence layer for structured technology). Second, your data loader is likely implementing a significant portion of the logic that is already encapsulated in your business objects. Your business objects will not be coded to fix problems in the legacy source data, but they will be coded to ensure consistency of your objects, including all referential integrity issues. The bottom line is that with this approach you are programming a lot of basic behavior in two places: in your business layer where it belongs and in your data loader where it does not. There has to be a better way.

7.2 Architected Data Loading

Figure 15 depicts an approach to data loading that is more in line with the needs of object development. The data loader application itself will be made up of a collection of classes. First, there may be several user interface classes, perhaps an administration screen for running the data load and a log display screen. Second, there will be a collection of business classes specific to the data loader, classes which encapsulate the data cleansing logic specific to the source data. You don't want this in your normal business classes because at some point your legacy source data is likely to go away and be replaced by the new and improved target data. There will also be classes that encapsulate the data load process logic itself, using the data load business classes to read the incoming data and then to create the “real” business objects for your application based on that data. If you are not doing a complete refresh of the target data you will need to first read the existing objects into memory, update them based on the source data, and then write them back out.

Copyright 1997-2005 Scott W. Ambler

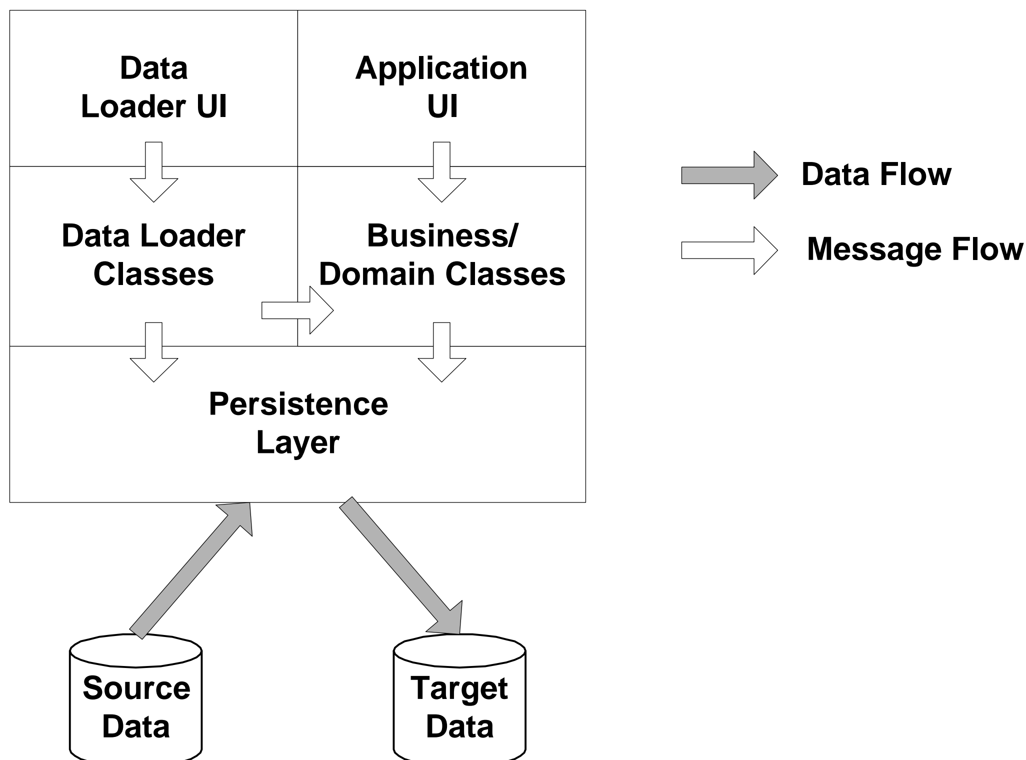

Figure 15. An architected approach to loading data.

There are two interesting points to be made about Figure 15. First, notice how your "data loader code" never directly accesses the source data – it goes through the persistence layer to get at the data. Second, the data loader code could easily be removed without affecting the applications and business classes, in other words the applications don't know and don't care about the source of the data that they manipulate.

There are several advantages to this approach:

? The data loader logic is decoupled from the schema for the target data, allowing you to update the target schema as needed by your business applications without requiring an update to your data loader.
? Key business logic is encapsulated in the business classes of your application, exactly where it belongs, enabling you to code it one place.
? Data cleansing logic is encapsulated in the business classes of your data loader, exactly where it belongs, enabling you to code it in one place.

There is one disadvantage to this approach: expensive data loading tools that your organization has purchased are likely not able to work within this architecture, likely based on the ancient/legacy approach of Figure 14, causing political problems for the users of those tools.

8. Supporting the Persistence Layer

How do you support this persistence layer within your organization? First, you need to develop an administration system that provides the ability to maintain instances of the mapping classes. This administration system would be updated by your persistence modelers responsible for developing and maintaining your persistence schema, and by your lead developers responsible for maintaining the object schema of your applications. You may also choose to add a cache to your persistence layer to improve its performance.

Copyright 1997-2005 Scott W. Ambler

To support the persistence layer an administration application needs to be built to maintain the instances of the **ClassMap** classes, as shown in Figure 16. These objects encapsulate the behavior needed to map objects into the persistence mechanism, including the complex relationships between the objects that make up your application, and form the information that is stored in the data dictionary for your application. This is the secret to a successful persistence layer: the objects stored in the data dictionary provide the behaviors needed to map objects into the persistence mechanism(s) where they are stored. When the design of your application or persistence mechanism schema changes you merely have to update the mapping objects within your data dictionary, you do not have to update your application source code.

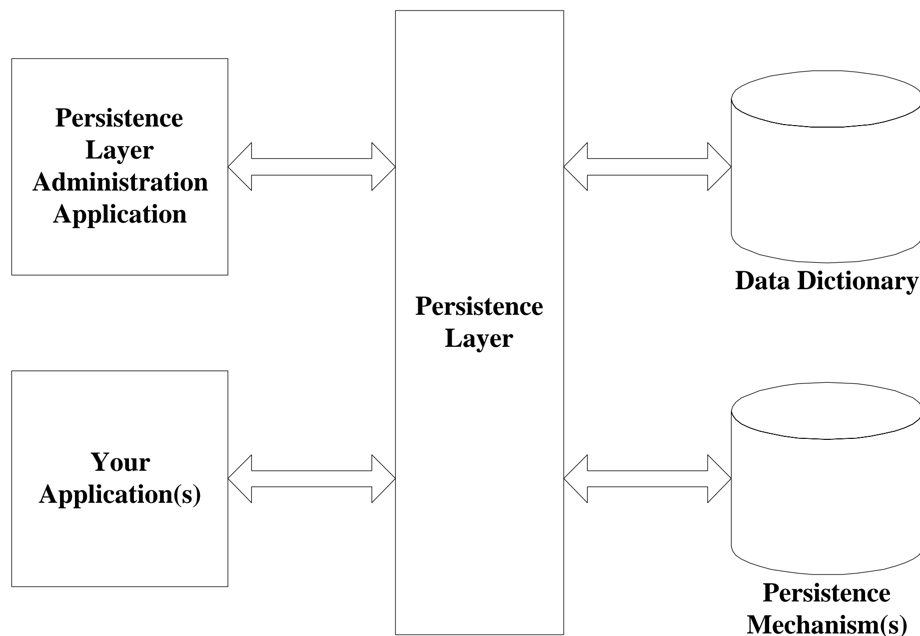

Figure 16. How the persistence mechanism works.

This approach to persistence effectively allows your database administrators (DBAs) to do what they do best, administer databases, without forcing them to worry about what their changes will do to existing applications. As long as they keep the data dictionary up-to-date they can make whatever changes they need to make to the persistence mechanism schema. Similarly, application programmers can refactor their objects without having to worry about updating the persistence mechanism schema because they can map the new versions of their classes to the existing schema. Naturally when new classes or attributes are added or removed to/from an application there will be a need for similar changes within the persistence mechanism schema.

Robust persistence layers protect application developers from changes made by database administrators and vice versa.

Copyright 1997-2005 Scott W. Ambler

9. Summary

The purpose of this white paper was to present a workable design for a robust persistence layer, a design proven in practice to work. It is possible for object-oriented applications to use relational databases as persistence mechanisms without requiring the use of embedded SQL in your application code which couples your object schema to your data schema. Technologies such as Java Database Connectivity (JDBC) and Microsoft's ActiveX Database Connectivity (ADO) can be wrapped using the design presented in this white paper, avoiding the inherent brittleness of applications whose design gives little thought to the maintenance and administration issues associated with persistence mechanisms. Persistence within object-oriented applications can be easy, but only if you choose to make it so.

10. About the Author

Scott W. Ambler is a senior consultant with Ambysoft Inc. Scott is the author of several books, including *The Object Primer 3rd Edition* (2004), *Agile Database Techniques* (2003), and *Agile Modeling* (2002). He has worked with OO technology since 1990 in various roles: Process Mentor, Business Architect, System Analyst, System Designer, Project Manager, Smalltalk Programmer, Java Programmer, and C++ Programmer. He has also been active in education and training as both a formal trainer and as an object mentor. Scott is a contributing editor with *Software Development* (www.sdmagazine.com). His home page is www.ambysoft.com/scottAmbler.html.

Copyright 1997-2005 Scott W. Ambler

11. References and Recommended Reading

Ambler, S.W. (1998a). *Building Object Applications That Work – Your Step-by-Step Handbook for Developing Robust Systems With Object Technology*. New York: SIGS Books/Cambridge University Press. http://www.ambysoft.com/buildingObjectApplications.html

Ambler, S. W. (1998b). *Process Patterns: Delivering Large-Scale Systems Using Object Technology*. New York: SIGS Books/Cambridge University Press. http://www.ambysoft.com/processPatterns.html

Ambler, S.W. (1998c). *Mapping Objects To Relational Databases: An AmbySoft Inc. White Paper*. http://www.ambysoft.com/essays/mappingObjects.html

Ambler, S.W. (1998d). *Persistence Layer Requirements*, Software Development, January 1998, p70-71.

Ambler, S.W. (1998e). *Robust Persistence Layers*, Software Development, February 1998, p73-75.

Ambler, S.W. (1998f). *Designing a Persistence Layer (Part 3 of 4)*, Software Development, March 1998, p68-72.

Ambler, S.W. (1998g). *Designing a Robust Persistence Layer (Part 4 of 4),* Software Development, April 1998, p73-75.

Ambler, S.W. (1998h). *Implementing an Object-Oriented Order Screen*, Software Development, June 1998, p69-72.

Ambler, S.W. & Constantine, L.L. (2000a). *The Unified Process Inception Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/inceptionPhase.html.

Ambler, S.W. & Constantine, L.L. (2000b). *The Unified Process Elaboration Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/elaborationPhase.html.

Ambler, S.W. & Constantine, L.L. (2000c). *The Unified Process Construction Phase*. Gilroy, CA: CMP Books. http://www.ambysoft.com/constructionPhase.html.

Ambler, S.W. (2004). *The Object Primer 3rd Edition: Agile Model Driven Development With UML 2*. New York: Cambridge University Press. http://www.ambysoft.com/theObjectPrimer.html.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *A Systems of Patterns: Pattern-Oriented Software Architecture*. New York: John Wiley & Sons Ltd.

Copyright 1997-2005 Scott W. Ambler

Index

A

Accessor methods 1
Administration application.............................. 23
AttributeMap 16

B

Buy vs. build.. 19

C

C++.. 5, 20
ClassMap 16
Class-type architecture...................................... 3
Client/server (C/S) architecture
  and class-type architecture 4
Concurrency.. 19
Connection... 6
Controller/process layer.................................... 4
Cursor 6, 9, 12

D

Data loading.. 21
Database administrators (DBAs) 7
DeleteCriteria.. 11
Design overview 8
Development schedule.................................... 21
Domain/business layer...................................... 4

E

Encapsulation.. 5
Extensibility.. 6

F

Feasibility study.. 19

H

HIGH/LOW approach 9

J

Java... 5, 20
Java Database Connectivity (JDBC).................. 7

L

Locking.. 19
  optimistic 19
  pessimistic.. 19

M

Modeling pattern
  layer 3

O

OID 6, 9
Open Database Connectivity (ODBC) 7
Optimistic locking... 19

P

Persistence layer 4
PersistenceBroker 14
PersistenceMechanism................................... 15
PersistentCriteria.. 10
PersistentObject 9
PersistentTransaction 13
Pessimistic locking.. 19
Proxy.. 6
ProxyAttributeMap 17

R

Record... 6
Referential integrity 22
Reflection.. 20
RelationalDatabase 15
Requirements 5
RetrieveCriteria... 11

S

Smalltalk 5, 20
SqlStatement 18
Structured Query Language (SQL) 1

T

Transaction 5, 9

U

UniDirectionAssociationMap 17
Unified Modeling Language (UML).................. 1
UpdateCriteria... 11
User-interface layer... 3

Copyright 1997-2005 Scott W. Ambler