A PERSISTENT OBJECT MANAGER FOR JAVA APPLICATIONS

By

ANURADHA SHENOY

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Stanley Su, for giving me an opportunity to work on this challenging topic and for providing motivation and continuous feedback during the course of this work and thesis writing.

I wish to thank Dr. Herman Lam for his guidance and support during this work. Thanks are due to Dr. Abdelsalam Helal for agreeing to be on my committee.

I would also like to thank Sharon Grant for making the Database Center a truly great place to work.

I would like to take this opportunity to thank my parents, my sister and Ashwin for their constant emotional support and encouragement throughout my academic career.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A PERSISTENT OBJECT MANAGER FOR JAVA APPLICATIONS
By

Anuradha Shenoy

August, 2001

Chairman: Dr. Stanley Y.W. Su
Cochairman: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

Object-oriented and object-based technology is becoming the predominant approach used to build flexible, scalable mainstream business software systems. Effective use of the object technology has demonstrated the ability to provide organizations with high-quality systems that are extremely understandable, maintainable and scaleable. Most business application programs need to deal with persistent data. Hence, a persistent object storage service is required to make these applications useful. Currently, three different methods of providing persistence exist: the programming language serialization approach, the Object-Oriented DBMS (OODBMS) approach and the Object-Relational Database Management System (ORDBMS) approach. Each approach has its advantages and disadvantages and needs to be chosen depending on the type of application.

This thesis presents the design and implementation of a light-weight persistent object manager (POM) for Java applications which are growing rapidly in popularity due to Java's platform independent and object-oriented nature. POM bridges the gap between

the Object-Oriented model and the relational model. It performs object-relational translation by devising its own set of standards. It allows object storage, retrieval, updation and deletion, in the form of APIs, for general-purpose applications written in Java. It also provides APIs customized to provide persistence to an Event Server application. The POM utilizes the underlying DBMS facilities to provide strong support for persistent data storage, management, backup, transaction, concurrency control, security, etc.

CHAPTER 1
INTRODUCTION

Object-oriented and object-based technology[1-3] is becoming the predominant approach used to build flexible, scalable mainstream business software systems. Effective use of the object technology has demonstrated the ability to provide organizations with high-quality systems that are extremely understandable, maintainable and scaleable. Most business application programs need to deal with persistent data. Hence, a persistent object storage service is required to make these applications useful.

We will begin by describing persistence and then looking at three different ways of providing persistence to object-oriented applications. Persistence is a term used to describe how objects utilize a secondary storage medium to maintain their state across discrete sessions of running applications. Persistence provides the ability for a user to save objects in one session and access them in a later session. When they are subsequently accessed, their states (e.g., attribute values) will be exactly the same as it was the previous session. However, in multi-user systems, this may not be the case since other users may access and modify the same objects.

The first form of providing persistence is using object serialization. Some object-oriented programming languages have an object serialization service. For example, the Java Object Serialization supports the encoding of objects and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream.[4] Serialization is used for lightweight persistence. A serialization file is created to store the encoded persistent objects at the runtime. After

the application terminates, the object instances still exist in the serialization files so that they can be reconstructed and brought into memory by any other running applications. This approach realizes the principle of object persistence for object-oriented applications. Also the data operations, including storing and retrieving, are provided by Java class library in the same Java programming language syntax. Therefore, the problem of impedance mismatch is avoided. One apparent drawback of the serialization service is the inability to support the management, sharing, and crash recovery for a large volume of data. It is, however, suitable for developing prototypes and simple applications when lightweight persistence and low developing costs are desired.

The second approach is OODBMS systems. The OODBMSs[5] also have their roots in object-oriented programming language (OOPL) and are designed to add persistence to objects that are used in OOPL, e.g., C++, Smalltalk and Java. OODBMSs solve impedance mismatch problems by providing extensive support for the data modeling features of one or more object-oriented languages. The data model that is used by an application is identical to the data model used by the OODBMS. From a programming language point of view, OODBMSs provide a seamless object persistence service. However, OODBMSs do not provide as good a query facility as ORDBMSs. Until recently, some of them did not provide a query language, or provided support for queries but with no support for query optimization and index management, thus rendering query support virtually useless for a large amount of data. The transaction rates supported by OODBMSs do not yet approach the high rates achieved by relational DBMSs using standard transaction processing benchmarks. Moreover, traditional OODBMSs provide little support for integrity constraints, security, view, and triggers.

In summary, OODBMSs provide a seamless object persistence service for the OOPL just like Serialization but with a greater support for managing, sharing and maintaining the reliability of a large volume of data. However, the performance of OODBMSs is still far behind that of RDBMSs and ORDBMSs.

The third approach to provide persistence is to use a proven, mature and stable technology like relational database management systems. Currently, relational database management systems are the most prevalent implementation of back-end physical data stores for business applications for the following reasons

- The relational model is simple and has a sound mathematical foundation.
- It is a mature technology that has been used and tested for a number of years and is consequently well understood.
- Organizations have made significant investments in the relational technology. They have purchased systems and software, trained their staff, and deployed mission-critical systems.
- Legacy systems contain critical corporate data, which resides in relational databases and must be used by new systems.

We have chosen a database management system called Cloudscape as our mechanism for persistent object storage and retrieval. This DBMS extends the functionality of a traditional RDBMS by providing a data type to store and process user-defined objects.

Object-orientation and the relational model are different paradigms of programming. When objects need to be stored in relational databases, the gap between the two different sights needs to be bridged. With full-blown object models the concepts

of object-oriented programming have to be mapped to relational table structures. These concepts are as follows:

- Aggregation
- Inheritance
- Associations between classes

During the conceptual phases of system development, an object-oriented design may resemble a relational design (e.g., the classes in the object model may be reflected in some fashion as tables or entities in the relational schema). However, as the design and implementation of the system matures, the goals of object-oriented and relational database development diverge. This divergence is caused because the objective of relational database development is to normalize data whereas the goal of object-oriented design is to model the problem domain as real-world objects. The result is an object-relational chasm. Hence, a mapping framework is required which bridges the gap between the object-oriented concepts and the relational database concepts. This framework must be strong and also flexible. This technical challenge motivated us to provide a light-weighted persistence service for object-oriented applications, called Persistent Object Manager (POM), by integrating the Java programming language, JDBC and the Cloudscape DBMS as shown in Figure 1.1.

The work in this thesis is bifurcated into two sections. The first section involves creating a general purpose POM. The second part is to customize it to provide persistence to an application called the Event Server.[6] The general purpose and the customized POM provide persistence services in the form of Application Programmers Interfaces (APIs).

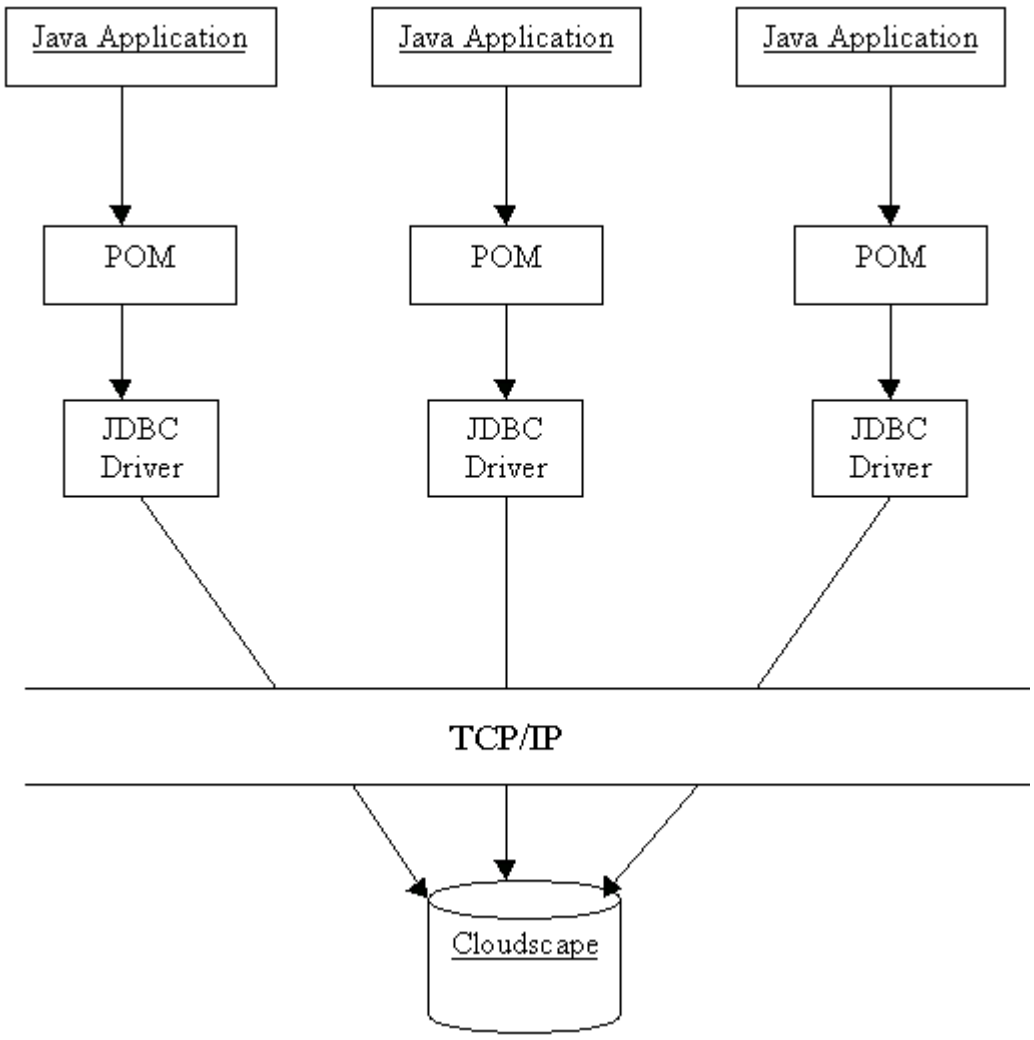

Figure 1.1 POM System Architecture

An overview of the features provided by the POM is given below.

- **Object storage**: An object structure could span over one or more relational tables. When an object needs to be stored in the database, the values of the appropriate attributes are extracted from the object and database specific structures are created with the attribute values (one or more SQL insert statements), and the structures are submitted to the database.

- **Object retrieval**: The process of object retrieval mirrors the process of object storage. When objects are restored from the database, data is retrieved from the database and translated into objects. This process involves extracting data from database specific structures retrieved from the data source, marshaling the data from database types into the appropriate object types and/or classes, creating the appropriate object, and setting the specific object attributes.
- **Object updation**: The POM provides two different mechanisms for updating objects. Objects can be updated either by passing an SQL update statement as a parameter to the POM's update API or by providing the object id of an existing object in the database and new values as parameters to the API.
- **Object deletion**: Object deletion can be done using two mechanisms, which are similar to the ones for updating objects. The user could pass an SQL delete statement as a parameter to the API for deletion. The other mechanism involves passing the object id of the object to be deleted. The POM ensures that objects that are deleted from an application system have their associated data deleted from the database.

Persistent object storage is of little use without a mechanism to search for and retrieve specific objects. Query facilities allow applications to interrogate and retrieve objects based on a variety of criteria. The POM uses a language called SQL-J, which allows the integration of Java syntax in the query formulation in order to extend the relational query model to make it object-centric rather than data centric.

All the features provided by the POM are executed as a transaction. The operations within a transaction are either executed entirely or not executed at all. The

POM also provides concurrency control and transaction management to enable data sharing among multiple users.

The remainder of this thesis is organized as follows. The next chapter describes the design and implementation details of the entire system. In Chapter 3 we take a look at the applications of POM, which include general-purpose applications and the Event Server application. And finally, Chapter 4 summarizes the presentation of the earlier chapters and discusses possible directions for future work.

CHAPTER 2
SYSTEM DESIGN AND IMPLEMENTATION

In this chapter we will look at the detailed design and implementation of the Persistent Object Manager. The POM has two main components namely, the general purpose Object-Relational Mapping Engine and the XML-Relational Mapping Engine for an Event Server application. The aim of the general purpose Object-Relational Mapping Engine is to provide a persistent storage facility for any Java application. It also provides a high-level mechanism, in the form of APIs, to store, retrieve, update and delete objects without worrying about the technicalities of how the objects are stored. The XML-Relational Mapping Engine provides persistence capability and a filtering mechanism to an application called Event Server. It includes a parser, which maps filters definitions stored in the form of XML files to a relational form and a set of Application Programmers Interfaces (APIs).

The POM has been implemented using an Object-Relational database called Cloudscape. Before we go into the intricacies of the design and implementation of POM, we will present an overview of the Cloudscape DBMS and the related software.

2.1 Cloudscape DBMS

Cloudscape[7] can broadly be described as a reliable, high performance Java embeddable database management system, implementing SQL-92 and requiring zero administration. It is written entirely in Java and can be deployed or used by other Java applications. Since it is an object-relational DBMS or ORDBMS, it provides some extensions to the traditional RDBMS functionality.

Below, we will look at the various methods of deploying the Cloudscape software.

- Embedded in a single-user Java application, Cloudscape can be practically invisible to the user, because it would require no administration and would run in the same Java virtual machine (JVM) as the application
- Embedded in a multi-user application such as a Web server, an application server, or a shared development environment.
- Embedded in a server framework called the "**RmiJdbcServer**" framework, which is provided with the Cloudscape software. This framework allows multiple client JDBC applications to connect to a central database server over a network.
- Deployed in a distributed system in which multiple, occasionally connected, applications have their own copies of the Cloudscape engine. Cloudscape can periodically connect to a central hub version of Cloudscape (called **Cloudsync**) to submit and receive updates.

In our implementation of the POM, we have chosen to deploy Cloudscape in the "**RmiJdbcServer**" framework mode so that the DBMS runs as a single server and all client Java applications connect to it across the network.

2.1.1 Serialize Datatype

In object-oriented programming an object consists of both data and corresponding behavior. Some RDBMSs can store complex objects as BLOBs (Binary Large Objects). But they cannot run any of the object's behavior or methods. But Cloudscape has a datatype called "**Serialize**", which can store and process user-defined objects in the database.

2.1.2 SQL-J

SQL-J[8] is the name of the Cloudscape's Java-enabled dialect of SQL. It stands for SQL for Java. Its syntax is a superset of the basic SQL-92 syntax. It allows the integration of Java syntax in the query formulation. It can be used to store objects in a database using the "Serialize" datatype and also execute methods that belong to those objects.

2.1.3 Cloudview

Cloudscape provides a graphical application that makes it easy to create tables or view the data in the database. This application is called Cloudview.[9] In this thesis we programmatically create tables at run-time and do not use the Cloudview tool. However, this tool can be used by client programmers to view the persistent data stored using the POM. It also provides an SQL window for executing SQL-J statements.

2.1.4 Advantages of Cloudscape

- Cloudscape is a lightweight database with a small footprint, which means that it doesn't use much memory or disk space, and is easy to administer.
- Since Cloudscape is written in Java, it can run in any environment that supports a JVM, or Java Virtual Machine.
- Cloudscape databases are portable across platforms and easy to manage.
- Cloudscape's Java extensions allow Cloudscape to store more than just SQL-92[10] data types. It can store instances of Java classes (objects).

2.1.5 Disadvantages of Cloudscape

- Currently, Cloudscape does not seem to have any feature for creating and managing users.
- Also, database security features are not very well developed yet.

2.2 Design of General Purpose POM

2.2.1 Overview of Object-Relational Mapping

Before we proceed with the details of the general-purpose POM system design, we will look at the concepts that are involved in an object-relational mapping. Traditionally, object-orientation and the relational model are different paradigms of programming and data representation. Therefore, when objects need to be stored in relational databases, the gap between the two needs to be bridged. The concepts of object-oriented programming like aggregation, inheritance and data types that are more complex than SQL data types need to be mapped to the relational table structures. In addition to that, some of the criteria discussed below needs to be taken into consideration during the mapping process.

- Performance

  This is one of the major forces that should be taken into account when mapping objects to tables. The mapping strategy has significant influence on the number of database accesses that occur in the system.

- Maintenance cost

  One of the goals of relational calculus is to eliminate redundancy using normal forms. However, relational database applications show best performance when the number of database accesses are minimal. Accesses to the database can be reduced by ignoring normal forms, which would have negative consequences on the maintainability of the application. We can clearly see that performance and maintenance are two conflicting goals. Hence, if the aim is to improve performance, the maintenance cost will be higher.

- Application type

  Applications, which are used to manipulate large sets of very complex interrelated objects, are not suited for using a relational database as a persistence mechanism. Some examples are CAD applications and CASE tools.

- Integration of legacy systems

  A number of times, applications have to be built on top of pre-existing legacy systems. In such cases, the mapping strategy should take the legacy data into consideration.

2.2.2 Concepts

We will now discuss some commonly used mapping strategies[11-13] including the strategies used in this thesis for mapping object-oriented concepts like inheritance and aggregation.

2.2.2.1 Inheritance

There are various ways to map inheritance hierarchies to relational database tables. Some of the patterns are presented below. This discussion does not cover multiple inheritance because our system is used by Java[14] applications and Java does not support multiple inheritance. Also, simple inheritance covers most practical cases.

Let us look at a sample inheritance scenario and consider the effects of different mapping strategies.

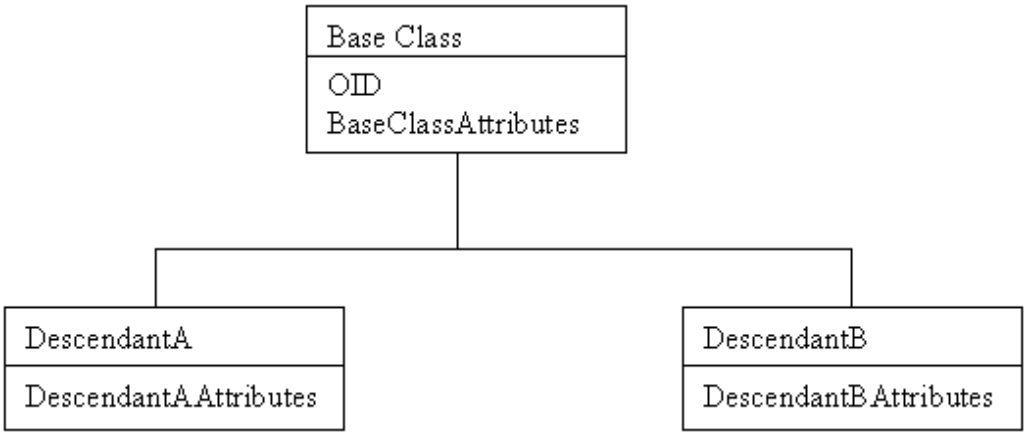

Figure 2.1 Inheritance Scenario

- Mapping Strategy 1-One Inheritance Tree One Table

This pattern demonstrates a way to map a complete inheritance hierarchy to a single database table. The solution that is normally suggested is to use the union of all the attributes of all the objects in the inheritance hierarchy as the columns of a single database table. Null values are used to fill the unused fields in each record. So, Figure 2.1 will be transformed as shown below:

Table 2.1 One Inheritance Tree One Table

	OID, BaseClass Attributes	OID, DescendentA Attributes	OID, DescendentB Attributes
BaseClass Instance	Attribute Values	Null Values	Null Values
DescendantA Instance	Attribute Values	Attribute Values	Null Values
DescendantB Instance	Attribute Values	Null Values	Attribute Values

Advantages:

- A single database operation is required to read or write.
- Mapping is straightforward and easy as long as the inheritance hierarchy does not become too deep.
- As the mapping is intuitively clear, formulating ad-hoc queries is fairly easy.

Disadvantages:

- The inheritance hierarchy cannot be changed once the mapping is done. This would mean that new classes derived from classes that are already mapped to the relational database cannot be added to the database. To overcome this problem, the entire table structure will have to be changed and the old instances should be reinserted with additional null values for the newly inserted columns. This is a very complicated procedure.
- Mapping too many classes to a single table will result in poor performance. Too much traffic on a single table can cause performance degradation and deadlocks.

- Mapping Strategy 2-One Inheritance Path One Table

This pattern demonstrates a way to map all attributes occurring in an inheritance path to a single database table. The solution that has been suggested to implement this approach is to map each class to a separate table. And in each table, store the attributes of all the superclasses in its inheritance hierarchy. This would transform Figure 2.1 to the following set of tables.

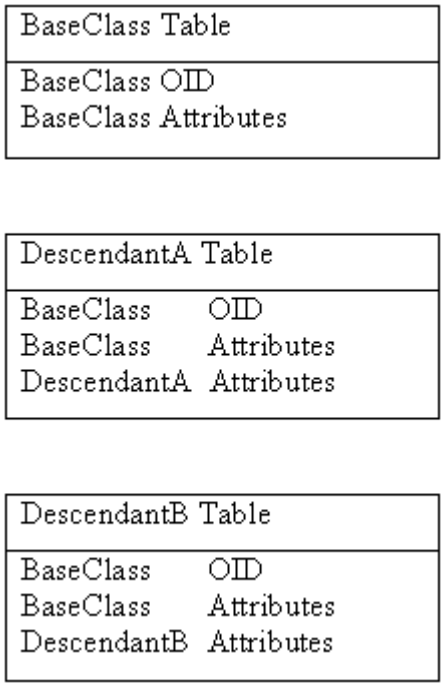

Figure 2.2 One Inheritance Path One Table

Advantages:

- The mapping needs one database operation to read or write an object
- There are no bottlenecks in tables near to the root of inheritance, because accessing an object exactly locks one table

Disadvantages:

- Adding or deleting attributes of a superclass results in changes to the tables of all derived classes.
- Since the base class attributes are stored in every derived class, there is redundancy in the information stored.
- As the mapping generally requires accessing more than one table to perform searches, ad-hoc queries are hard to write for inexperienced users.

- Mapping Strategy 3-Storing Objects in BLOBs

BLOB (Binary Large Object) is a new SQL3 datatype, which gives a relational database more flexibility in what can be used as a type for a table column. A BLOB can store very large amounts of data as raw bytes. In this mapping pattern we will discuss a way to map objects to a single database table using BLOBs. The general way to map would be to create a table with two columns for each class. One column would store the object id OID (instance identifier), and the other column would store the object instance in a BLOB. In this case, Figure 2.1 would transform as follows.

BaseClass OID	BLOB to store BaseClass objects

DescendantA OID	BLOB to store DescendantA objects

DescendantB OID	BLOB to store DescendantB objects

Figure 2.3 Storing Objects in BLOBs

Advantages:

- Objects in BLOBs allow reading and writing of any object in one database operation.
  - If the database allows variable length BLOBs, space consumption is optimal.
  - Schema evolution is comparable to schema evolution in an object-oriented database.

Disadvantages:

- The internal structure of the BLOB is not accessible easily. Hence, separate functions that give access to the attributes have to be registered with the database. Also, defining and maintaining these functions can prove to be very costly.
- As scanning classes for properties is difficult, ad-hoc queries are difficult to express.

- Mapping Strategy 4-Our Approach

The main aim of the Persistent Object Manager is to provide persistence and high-level querying to Internet based applications like the Negotiation Server, the Event Server, etc. These applications can have inheritance hierarchies that are very deep and are continuously growing. On review of all the above-mentioned mapping strategies for inheritance, we notice that the disadvantages are far too overwhelming as compared to the advantages. Therefore, none of the above mentioned mapping strategies are suitable for such applications.

Now, we will look at the design of our mapping strategy that uses some concepts from a few of these mapping strategies but also tries to overcome most of the disadvantages mentioned above. Our approach is to map each class in an inheritance hierarchy to a separate database table with a superclass OID column to link the rows in each derived class table with the corresponding rows in the parent class table. This would lead to the following mapping for the inheritance hierarchy mentioned in Figure 2.1.

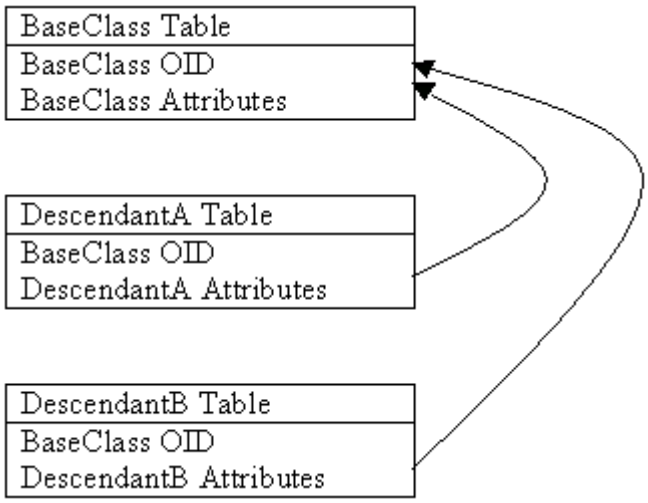

Figure 2.4 POM approach for mapping inheritance

Advantages:

- This pattern provides a very flexible mapping. The inheritance hierarchy need not be fixed. Increasing the depth of the inheritance hierarchies can be easily performed without altering any of the pre-existing tables.
  - The mapping has near optimal space consumption. The only redundant attributes are the base class OIDs, which are needed to link the levels of hierarchy.
  - As the mapping is straightforward and easy to understand, schema evolution is straightforward. This reduces the burden of maintenance.

Disadvantages:

- The main disadvantage of this strategy is the formulation of queries. Since the mapping could require accessing more than one table to retrieve an object instance's data, queries are hard to formulate for inexperienced users. To overcome this drawback, a high-level query processor has been created, which allows the user to query the database without worrying about the depth of the

inheritance and the number of superclass tables. The user interacts with the query processor through a set of APIs provided by the POM.

2.2.2.2 Aggregation

The aggregation relationship in an object-oriented data model is simply that a class consists of a set of attributes and the domain of an attribute may be any class. This is one of the main departures from a relational database model where the domain of an attribute is restricted to be of a primitive type. Below, we will define a sample object model and use it to discuss some of the common strategies for mapping aggregation.

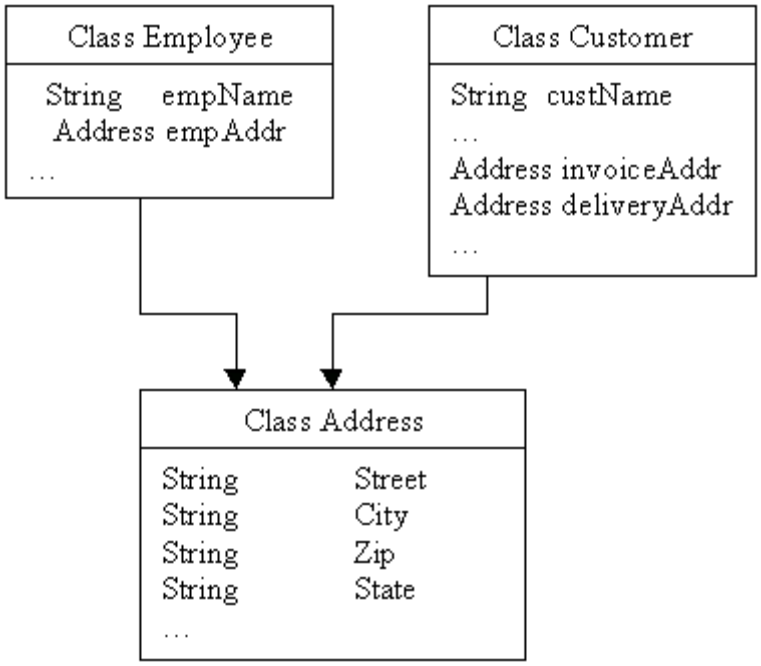

Figure 2.5 Aggregation Scenario

- Mapping Strategy 1-Single Table Aggregation

  This pattern suggests that aggregation can be mapped to a relational data model by integrating all aggregated objects attributes into a single table. When applied to the sample scenario in Figure 2.5, the resulting mapping would be as shown below.

Employee	Table
EmpName	VARCHAR(20)
…	
EmpStreet	VARCHAR(30)
EmpCity	VARCHAR(30)
EmpZip	VARCHAR(30)
EmpState	VARCHAR(30)

Customer	Table
CustName	VARCHAR(20)
…	
InvStreet	VARCHAR(30)
InvCity	VARCHAR(30)
InvZip	VARCHAR(30)
InvState	VARCHAR(30)
…	
DelStreet	VARCHAR(30)
DelCity	VARCHAR(30)
DelZip	VARCHAR(30)
DelState	VARCHAR(30)

Figure 2.6 Single Table Aggregation

Advantages:

- This solution is optimal in terms of performance, as only one table needs to be accessed to retrieve an object.
- Aggregated objects are automatically deleted on the deletion of the aggregating objects.

Disadvantages:

- Change in structure of the aggregated object can prove to be a problem during maintenance.
- Mapping Strategy 2-Foreign Key Aggregation

According to this strategy, the aggregating object is mapped to a table and the aggregated object is mapped to another table. The aggregating object's table stores

the aggregated object's OID, which serves as a foreign key link. If we apply this solution to the object model in Figure 2.5, we get the following mapping.

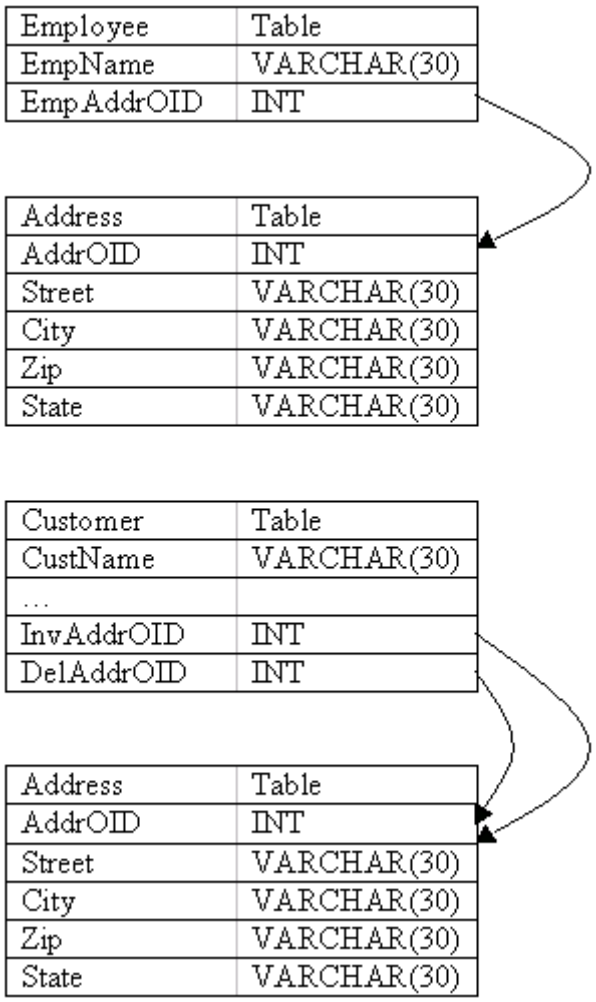

Figure 2.7 Foreign Key Aggregation

Advantages:

- Mapping is more flexible and easy to maintain.

Disadvantages:

- Foreign key aggregation needs a join operation, which is at least two database accesses. This reduces the performance factor.

- Since aggregated objects are not automatically deleted on the deletion of aggregating objects, database level triggers must be provided for this purpose.
- Mapping Strategy 3-Our Approach:

  The POM is implemented on top of the Object-Relational database called Cloudscape. In addition to the primitive SQL data types, Cloudscape supports a data type called "Serialize". This data type allows complex Java objects to be stored as columns in a relational table. However, to be storable, the object class should implement the "java.io.Serializable" interface, should be declared public and be available to the Java Virtual Machine (JVM) in which it is running. Like we mentioned in Chapter 1, *Serialization* is a Java mechanism for reading and writing objects to a stream. When an object is serializable, Java knows how to write the value of each of its fields to an output stream and how to read the value of each of its fields from an input stream.

  In this thesis, we have used an approach similar to the Single Table approach for mapping aggregation. However, since the DBMS we have chosen provides a feature for storing complex objects, the POM stores the aggregated object as a column in the relational table for the aggregating object's class. When our approach is applied to the sample scenario in Figure 2.5, the resulting mapping is as shown below.

Employee	Table
EmpName	VARCHAR(20)
…	
EmpAddr	Serialize(Address)

Customer	Table
CustName	VARCHAR(20)
…	
InvAddr	Serialize(Address)
…	
DelAddr	Serialize(Address)

Figure 2.8 POM Approach for Mapping Aggregation

Advantages:

In addition to the advantages of the Single Table approach, our approach has the following advantage

- The Cloudscape DBMS allows the user to formulate queries, which execute get methods on the attributes of the "Serialize" data type. An example of such a query would be

  Select * from Customer where InvAddr.getState()='FL'

Disadvantages:

- Serialization is a relatively slow process. It is more efficient to store built-in relational data types than objects. Hence, we use the Serialize data type only in the case of storing aggregated objects.

2.2.3 Classes with Dependent Attributes

As we described in Section 2.1.2.2, the default approach used by the POM to map aggregation is to store the aggregated object as a serialized column in the database. However, there could be some cases where a class could have a many-to-many relationship between its attributes. If these attributes are aggregate objects, there would

be redundancy when storing the objects. In such cases, it would be ideal to store the aggregated object in the relational table corresponding to its parent class and maintain a foreign key reference to the object in the relational equivalent of the class with the many-to-many relationship. In order to facilitate this, we have designed a GUI to identify classes with dependent attributes.

2.3 Design of POM for Event Server

2.3.1 Background of Event Server Framework

In this section, we will look at the background of the Event Server framework.[3] It is a general framework, which extends the current Web model by allowing information providers and software or application system owners to publish their information resources and system services in a uniform and structured way. In this framework, each information provider (*publisher*) will model his/her information resources or application systems in terms of not only attributes and methods but also of events, knowledge rules, and triggers. The users of an information resource or application system, through browsers and/or search engines, can access not only the information resource or the service of the application system but also its meta-information. They can subscribe to the events associated with it and tie some rules to the events by some trigger specifications. When an event is posted (e.g., an update of the information resource or an activation of a service provided by the application system at the provider's side), the knowledge rules specified by the provider will be automatically triggered for processing by an Event-Trigger-Rule server running as a plug-in to the provider's Web server. The event notification conditions of the subscribers of the event will then be verified. Those subscribers, whose notification conditions are satisfied, will be notified. The notification

will trigger the processing of users' knowledge rules by the Event-Trigger-Rule servers running as plug-ins to the users' Web servers.

The architectural components of this framework are divided into several logical subgroups, depending on their functionality and the activities they perform at both run-time and build-time. The subgroups are strongly interconnected and have shared components.

The first group, *Event and Filter Definition Group*, covers build-time activities on the publisher site. A GUI for event/filter design is used for defining and editing events and filters. Once defined and edited, XML definitions for the corresponding events and filters are generated. At the same time, the events are translated into corresponding Java classes. The generated XML files and the Java event classes are installed in shared directories, which are also known and accessible to the event-posting generator component and the POM. During run-time, the generated event classes are used by the *Supplier* (event posting generator) component to send events.

The second group, *Event Registration and Filter Settings Group,* covers one of the run-time activities, which is user registration for the occurrence of a certain event. After the user enters his/her preferences (i.e., data condition that should be satisfied as a precondition of an event notification), the Registration Servlet stores the client profile in a persistent store (database) by using the POM.

The third group, *Event Posting Group*, also covers one of the run-time activities. Event Posting Generator components (DBMS, Java application, Monitor) are responsible for explicitly posting the event but are not concerned with how the event is handled further.

The fourth group, *Event Filtering and Rule Server Group*, matches the posted events against the stored user profiles, by using a *Profile Manager*. The Profile Manager regulates the dispatching of events by using the data stored in the *Persistent Storage (database)*, which is accessed by the POM middleware.

The fifth group, *Event Notification Group*, sends the notification through the Internet and stores the event notifications into the User Profiles Persistent Storage.

The sixth group, *Personalized system and Event-Rule Management Group*, allows the user to personalize the display of the requested information.

2.3.1.1 Event filters

Every event has an associated event filter with one or more attributes. There are two special types of attribute filters namely, "*Single Selection List*" and "*Multiple Selection List*". The "*Single Selection List*" on an attribute allows the user (subscriber) to select one of the many predefined values provided for that attribute. The "*Multiple Selection List*" on an attribute allows the user to select one or more values from a list of values provided for that attribute. The comparison operators are "*Equality*" and "*Range*". The range operator is used to define an ordered pair of minimum and maximum integer values over some attributes. Figure 2.9 shows an example of an event filter for *NumismaticOffer* event, defined in XML. This event filter consists of three attributes: *Coin type*, *Quality*, and *Years*.

```
<EventFilter>
<SingleSelectionList>
< AtrrName> Coin Type </AttrName>
<Type> String </Type>
<ListValues> Silver Dollars, Silver
Commemorative, Half-Dollar
</ ListValues>
</SingleSelectionList>
<AttrFilter>
<FilterNum> 1 </FilterNum>
< AtrrName> Quality </AttrName>
<Type> String </Type>
</AttrFilter>
<MultipleSelectionList>
< AtrrName> Years </AttrName>
<Type> Integer </Type>
<ListValues> 1885, 1890, 1907 </ ListValues>
</MultipleSelectionList>
</ EventFilter >
```

Figure 2.9 Sample Filter

A subscriber registers for this event by providing values for these attributes using the registration Servlet. An example is shown in Figure 2.10. The displayed information, along with the subscriber information is the subscriber's *profile*. This profile is stored persistently in the database, using the POM. When the event is posted, the Event Server queries the POM to get the list of subscribers who have registered for the event and whose event notification conditions match those of the posted event.

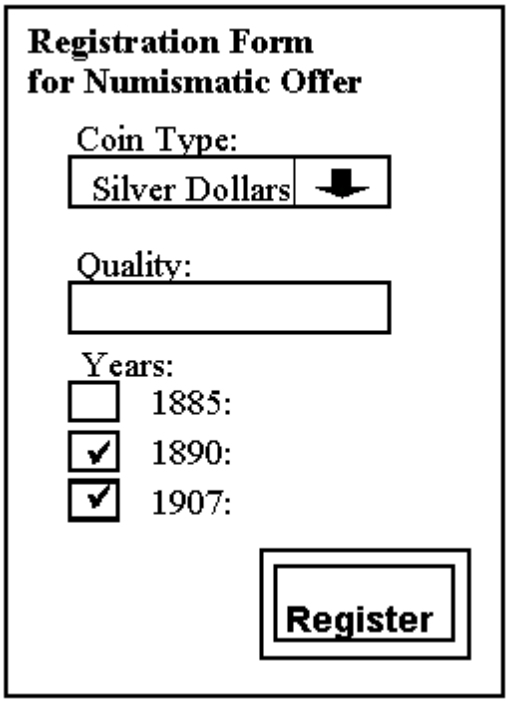

Figure 2.10 Registration Servlet

2.3.1.2 XML - Relational mapping

Since the event filters are in the form of XML files, an XML-Relational parser has been developed to convert the XML document into a corresponding relational table. The parser maps each filter definition to one or more relational tables. All simple filter attributes, including "*Single Selection List*" are stored in the primary filter table. Each attribute of type "*Multiple Selection List*" is treated as a multivalued attribute. Hence for each multivalued attribute, a separate table, which references the primary filter table using the filter id, is created. If an attribute *attr* has the range operation specified in the filter definition, two columns, *attr_min* and *attr_max* are created in the relational table.

2.3.2 Event Registration and Notification

The POM provides services to the *Event Registration and Filter Settings Group* and the *Event Filtering and Rule Server Group* in the form of Application Programmers Interfaces (APIs).

2.3.2.1 Previous mechanism

In the previous mechanism, the subscriber registration profile is serialized and stored on the disk. When an event is posted, the profiles are loaded into memory and are filtered in order to find all subscribers who have registered for the event. To reduce the number of profiles that must be examined, an inverted index of profiles is built, in order to speed up the filtering process. Hence, the foundation of the profile-matching algorithms is a compact data structure for representing a large collection of profiles called *Profile Index*.[3] However, the inverted index lists are not suitable for range queries. Therefore, a modified version of 2-3 Trees (or Red-Black Trees) is used to solve the range search problem.

2.3.2.2 POM mechanism

As we can see from Section 2.3.2.1, the previously used mechanism stores the user profiles as serialized files on disk. When an event is posted, these profiles are brought into memory and stored in complex data structures. Also, the filtering process is performed by executing complex algorithms. The POM on the other hand, performs an XML-Relational mapping and stores the subscriber profiles in the database. During the filtering process, it generates dynamic queries to retrieve all subscribers whose notification conditions have been satisfied. The queries are formulated by taking into consideration the existence of specialized filter attributes like "*Single Selection List*" and "*Multiple Selection List*" and operators like the "*Range*" operator. This mechanism is faster and more efficient than the previous filtering mechanism. In addition to this, the POM also provides RDBMS features like transaction management, concurrency control and crash recovery.

2.4 Implementation of the general-purpose POM

In this section, we will look at the implementation details of the general-purpose POM. The general purpose POM is mainly intended to provide persistence capability to applications written in Java. In order to provide persistence, the class should extend the *java.io.Serializable*[1] interface. In addition to that, the class should provide **get** and **set** methods for all its attributes. The POM uses Java Reflection APIs when storing and retrieving objects. Reflection enables Java programs to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions.[14] With the Java Reflection API,[15] the application program can get and set the value of an object's attribute, even if the attribute name is unknown to the program until run-time, and invoke a method upon an object, even if the method is not known until run-time.

2.4.1 General-Purpose APIs

The general-purpose POM provides the following features in the form of APIs

- Get RMI Connection

  public void getRMIConnection()

  This API gets the RMI connection to the Cloudscape database server. The parameters for the connection should be specified in a configuration file (Config.txt), in the directory in which Event Service classes are stored. The parameters include

  - Host computer on which the Cloudscape database server is running.
  - The connection port.
  - The name of the database (*db*).

- Insert Object

  This API allows the user to insert an object in the database. The API checks if a relational table exists for the class. If there is no relational equivalent for the given class definition, one or more tables are created based on the POM's mapping strategies that we described in Section 2.1. In case of inheritance, the extended class maintains a reference to the base class. This information is stored in a "*FOREIGN_KEYS*" table. In case of aggregations, the aggregated object is Serialized and embedded in the table as a column. The interface to this API is given below:

  public void insertObject(String className, Object obj)

- Query Objects

  This API is used to query objects, which satisfy a certain condition, from the database. It accepts a query string as a parameter. It modifies the query string in cases of inheritance, by adding all the inherited classes to the FROM clause and the necessary JOIN conditions. The query is then executed at the backend. The result is returned in a Java data type called ResultSet, which is similar to a relational table. The data in each row is used to construct an object. The set of objects and their corresponding object ids (OIDs) is returned in a result vector. The interface to this API is given below:

  public Vector queryObjects(String query)

- Update Object

  This API is used to update objects in the database. It has been implemented with the following two interfaces

  - *public void updateObject(int OID, Object obj)*
  - *public void updateObject(String className, String updateStatement)*

The first interface can be used in applications, which use a graphical user interface (GUI) to interact with the persistent objects. In such cases, the user would query the database to obtain the object and its object id, update it using the GUI and use the *UpdateObject* API to make the changes persistent.

The second interface can be used by applications, which need to update one or more objects, which satisfy a certain condition. For example, in a University database, the user could pass an update query such as "Update Student Set gpa=gpa*1.1 where ssn=123" to this API.

- Delete Object

This API is used to delete one or more objects from the database. Like the updateObject API, this API is also implemented with two interfaces.

- *public void deleteObject(String className, int OID)*
- *public void deleteObject(String className, String deleteStatement)*

The first interface can be used in applications, which use a graphical user interface (GUI) to interact with the persistent objects. In such cases, the user would query the database to obtain one or more objects and use the deleteObject API to delete a particular object after ascertaining that it meets the deletion criteria.

The second interface can be used by applications, which need to delete one or more objects, which satisfy a certain condition. For example, in a University database, the user could pass an update query such as "Delete Student where year='Senior'" to this API.

This API will delete all objects that inherit from it and also all objects from which it inherited. For example, if the inheritance hierarchy is of the form Person → Student →

GraduateStudent, and the *deleteObject* API is used on Student, it will delete the corresponding instances from all three classes.

- Insert Instance

  This API is used to insert an object, which inherits from an object that already exists in the database. The main objective behind using this API is to prevent storing redundant objects. The interface to this API is of the form:

  public void InsertInstance(Object obj, String condition)

  The "*condition*" is used to link the object to its predecessor.

  For example, if Class Student extends Class Person, the user could create an instance of Student by specifying values for only the Student specific attributes and call this API. The "*condition*" could be of the form "SSN=12345". The API ensures that all objects in an inheritance hierarchy share the same object id.

- Remove Instance

  public void removeInstance(String className, int OID)

  This API is used to remove a particular instance of an object from the database. It deletes a particular object and all objects, which inherit from it. However, unlike the *deleteObject* API it does not delete the objects it inherits from.

  For example, if the inheritance hierarchy is of the form Person → Student → GraduateStudent, a call to this API with a Student object as parameter will result in the deletion of the Student object and the corresponding GraduateStudent object. However, the corresponding Person instance continues to exist in the database.

2.4.2 GUI for Identifying Dependent Classes

A snapshot of the GUI for identifying dependent classes is shown in Figure 2.11.

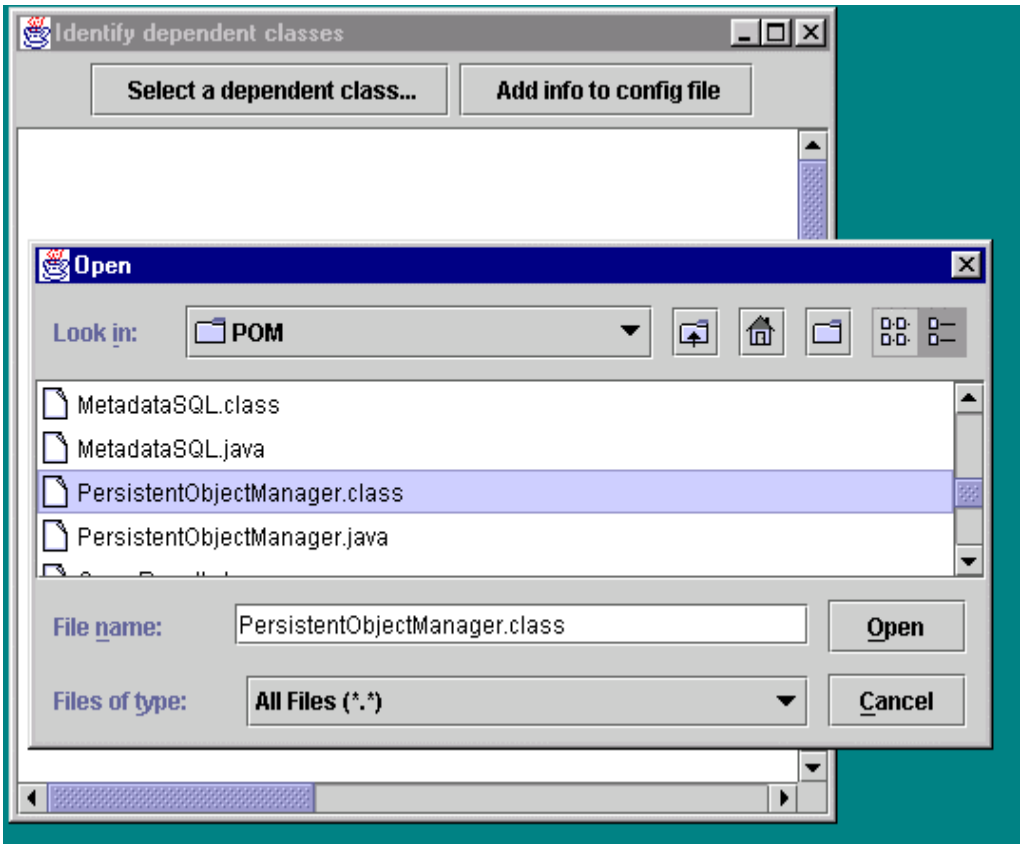

Figure 2.11 GUI for Identifying Dependent Classes

The classes, which are identified to have many-to-many relationship, are stored in a file and are the aggregated objects in the class are mapped with foreign key reference to the relational equivalent of the aggregated object's class.

2.5 Implementation details of POM for Event Server

2.5.1 Build-Time Activities

The *Event and Filter Definition Group* of the Event Server framework creates XML definitions for events and filters, which are stored in a shared directory, which is known and accessible to the POM. The POM uses a configuration file which stores information about the underlying database name, the host computer on which the

database resides, the port on which the POM RMI server is running and the absolute path of the shared directory in which the XML definitions of events and filters are stored.

Sample of configuration file

host madrid

port 2099

database POMDB

dir R:\ETR\Isee\NewESP\POM\XML\

2.5.2 Run-time Activities

At run-time the POM provides service to the *Event Registration and Filter Settings Group* and the *Event Filtering and Rule Server Group* in the form of APIs. Below, we will look at each API in some level of detail. In Chapter 3 we will look at sample scenarios to demonstrate the use of each API.

1. public void getRMIConnection()

This API gets the RMI connection to the Cloudscape database server. The parameters for the connection should be specified in a configuration file (Config.txt), in the directory in which Event Service classes are stored. The parameters include

- Host computer on which the Cloudscape database server is running.
- The connection port.
- The name of the database (*db*).
- The absolute path of the directory in which the XML files for the filter definitions are be stored.

2. public void createSubscriberTable()

This API is required to create a SUBSCRIBER table. When a subscriber registers for an event, the subscriber profile will be stored in this table. Since this

table has fixed attributes, it can be created when the EventManager is started. When the API is executed, it checks the database (*db*) if the SUBSCRIBER table already exists. If it does not exist, this API will create a SUBSCRIBER table with the following fields

- SubscriberName
- SubscriberIPAddress
- SubscriberEmailAddress
- EventName
- FilterName
- FilterId

The combination of event name, filter name and filter id will be used when querying for subscribers who have subscribed to a particular event.

3. public void createEventInfoTable()

This API is required to create an EVENTINFO table. This table stores all unique event names in the system. When the Distributor crashes and recovers, the data in the EVENTINFO table will be used by the Distributor to reconstruct the Distributor table. This API should be executed once when the EventManager is started. It checks the database (*db*) if the EVENTINFO table already exists. If it does not exist, this API will create an EVENTINFO table with the following fields

- EventName

4. public void storeSubscriberProfile(String subName, String subscriberIPAddr, String subEmailAddress, String eventName, String filterName, Vector formControlsParams)

When a subscriber registers for an event, this API is used to store the subscriber's profile in the database. The parameters to this API are

- The subscriber name.
- The subscriber's IP address.
- The subscriber's e-mail address.
- The name of the event the subscriber has registered for.
- The name of the filter template for the subscribed event.
- Vector containing the values for the filter attributes (event notification parameters).

This API checks if a relational table for the given filter template exists in the database (*db*). If a table does not exist, it creates a table by parsing the XML file "*filterName.xml*" which will be stored in the shared directory specified in the Configuration file. The XML parser also takes care of multivalued attributes by creating separate tables for each attribute, which is of the type "*MultipleSelection*". These tables reference the filter id of the parent table.

After performing the XML-Relational mapping, the API dynamically generates a set of SQL-J statements to store the subscriber's profile in the Subscriber table and the corresponding filter table(s). It also stores the event name in the EVENTINFO table if it does not already exist in the EVENTINFO table.

5. public Vector getSubscriberOfEvent(Object event)

When an event is posted on the provider's side, all subscribers, whose event notification conditions are satisfied, need to be notified of the occurrence of that

event. This API basically performs the filtering of the subscriber profiles by accepting an event object as a parameter and verifying the event notification conditions of all the subscribers of that event. It uses Java Reflection APIs to access the values of the attributes of the given event object. These values are then used to generate one or more SQL statements to obtain the list of subscribers whose event notification criteria, match the values of the posted event. Multiple SQL statements are created only if there are multivalued attributes in the filter for that event. The list of subscribers is returned in a result vector.

6. ***public void updateSubscriberProfile(String subName, String subscriberIPAddr, String subEmailAddr, String eventName, String filterName, Vector formControlsParams)***

This API is used when a subscriber of an event wishes to update his/her notification criteria. It updates the filter attribute values for the given subscriber with new values, which are stored in the formControlParams vector. The filter table to be updated is obtained from the filter name. Multiple tables may have to be updated in the case of multivalued attributes.

7. ***public void deleteSubscriber(String subName, String subIP, String subEmail, String eventName, String filterName)***

This API is used when a subscriber of an event wishes to unsubscribe for that particular event. This API deletes the given subscriber from the SUBSCRIBER table. It also deletes the filter information for this subscriber from the corresponding filter table(s).

8. public void deleteEvent(String eventName)

This API is used when the publisher of an event wishes to delete that event. The API internally deletes all subscribers subscribed to the given event. It also drops all filter tables associated with the event.

9. public Vector getEvents()

This API is used to reconstruct part of the distributor table on crash recovery of the Distributor. It returns the set of unique eventNames, which are currently in the database. This information is obtained from querying the EVENTINFO table.

10. public int countSubscribersOfEvent(String eventName)

This API is used to reconstruct part of the distributor table on crash recovery of the Distributor. It returns the count of the subscribers subscribed to the given event. This information is obtained from querying the SUBSCRIBER table.

CHAPTER 3
SAMPLE SCENARIOS

In this chapter we will look at some sample scenarios to demonstrate the use of the features provided by the general-purpose POM and the APIs customized for the Event Server application.

3.1 Scenario for General Purpose POM

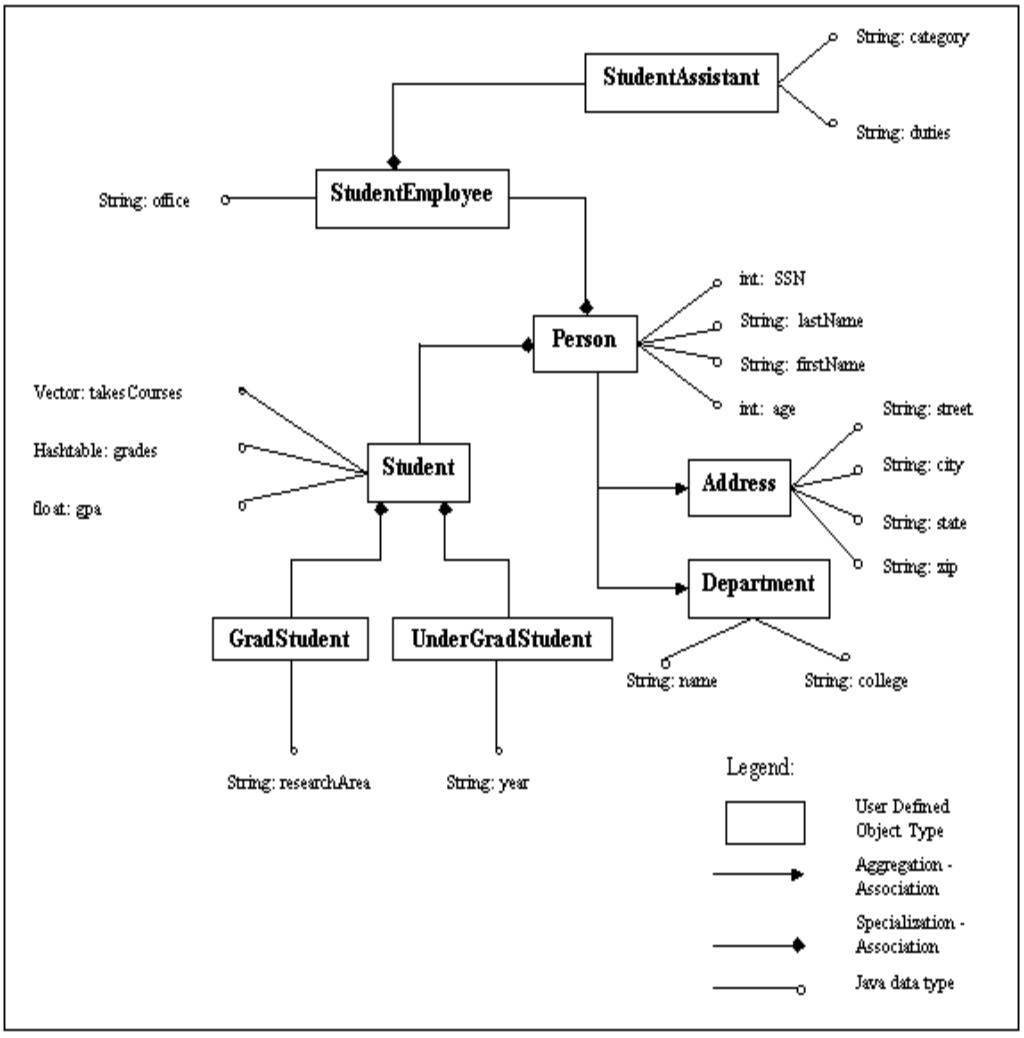

Figure 3.1 A Schema Graph for a Sample University Database

We have created a university database scenario above to demonstrate all the features provided by the general-purpose POM. This scenario provides examples of object-oriented concepts like inheritance and aggregation and also includes complex Java data types like Vector and HashTable.

3.1.1 Hierarchy of Classes

In this section, we will look at the hierarchy of the classes defined in our scenario. The class **Person** is at the root of the inheritance hierarchy. In addition to primitive data types, it has aggregated data types of classes **Address** and **Department**. The classes **Student** and **StudentEmployee** inherit from the class **Person**. The classes **GradStudent** and **UnderGradStudent** extends the class Student; and the class **StudentAssistant** inherits from the class **StudentEmployee**. The member attributes of each of these classes are either Java data types or aggregated objects.

3.1.2 General Purpose API Usage

In Chapter 2 we have reviewed the design of the POM APIs. In this section, we will demonstrate the usage of the APIs that can be used by general-purpose applications.

- InsertObject:

  This API is used to store objects in the persistent store. An object of a class is created by specifying values for all its member attributes, including those inherited from other classes. The object is then made persistent by making a call to this API. Each object will have a unique OID assigned to it. It is important to note that the OID is common to all objects in an inheritance tree.

  For example, we will create a new object of class **Person** (*person*)**,** with SSN=123 and insert it into the database as shown below

  Person *person* = new Person();

person.ssn = 123;

person.lastName = “Lee”;

…

insertObject(*person*)

Similarly, we will insert a new object of class **Student** (*student*), with SSN 456, by specifying values for all its attributes including those inherited from class **Person**.

- QueryObjects:

  This API can be used to query objects of a particular class, which satisfy a certain condition. An example is shown below

  String query = ”Select * from Person where address.getCity()=’Gainesville’”;

  Vector result = queryObjects(“Person”, query);

  The resulting vector will contain the objects and their corresponding OIDs.

- InsertInstance:

  This API is used when an instance of a specialized class needs to be made persistent. It is linked to its superclasses by a unique user-defined identifier. In the university database scenario, the SSN is a unique identifier. For example, an instance of class **Student** (*student*), *can* be created by specifying values for its declared attributes. This instance could be made persistent by linking it to an already existing object of the class **Person** (e.g., SSN=123). Below, we show the usage of this API.

  insertInstance(stud, “SSN=123”);

  To demonstrate the usage of the *RemoveInstance* and *DeleteObject* APIs, we will create instances of class **GradStudent**, **StudentEmployee** and **StudentAssistant** and

use this API to link them to SSN 123. We will also create an instance of **UnderGradStudent** and link it to SSN 456.

- UpdateObject:

  This API is used to update an object after executing a query. It is also used to update all instances, which satisfy a given condition. Examples for usage of both the APIs are shown below.

  - String query = ”Select * from Person where address.getCity()=’Gainesville’”;

    Vector result = queryObjects(“Person”, query);

    QueryResult qRes = (QueryResult) result.elementAt(0);

    int OID = qRes.oid;

    Person person = qRes.result;

    // Update the person’s attributes and call the API

    updateObject(OID, person);

  - String update = “Update Person set age=age+1 where lastName=’Lee’;

    updateObject(“Person”,updateStmt);

- RemoveInstance:

  This API will remove an instance and all instances that inherit from it, from the database. In our example scenario, if we execute this API on a student with SSN=123, it would remove the corresponding **Student** and **GradStudent**. However, the **Person**, **StudentEmployee** and **StudentAssistant** with SSN=123 would continue to exist in the database, because they form a different path in the inheritance tree.

  The API can be used after performing a query because it requires the OID of the instance to be removed. The user should query the database using the

queryObjects API, which returns the object and its corresponding OID. Below, we will demonstrate the usage of this API.

- String query = ”Select * from Student where SSN=123”;

  Vector result = queryObjects(“Student”, query);

  QueryResult qRes = (QueryResult) result.elementAt(0);

  int OID = qRes.oid;

  removeInstance(OID, “Student”);

- DeleteObject:

  This API is used to delete an object completely from the persistent store. In addition to deleting the given object and all objects that inherit from it, it deletes all the objects higher up in the inheritance hierarchy also. In order to call this API, one would need to specify the OID of the object.

  For example, in our sample scenario, if we use this API to delete the **Student** with SSN=456, it would remove the **Person**, **Student** and **UnderGradStudent** with SSN=456. Basically, all instances in the inheritance hierarchy that are related to the given object are deleted.

3.2 POM for Event Server

In this section, we will look at a sample scenario to demonstrate the usage of the APIs intended for the Event Server application. We will assume that the Event Manager has been started, and a connection to the database is already established using the POM. At this time, the POM will have created the **Subscriber** table using the ‘*createSubscriberTable*’ API, if it did not already exist in the database.

Let us assume that an event called *SpecialTicket* with member attributes, '*FromCity*', '*ToCity*' and '*Price*' is created by a certain publisher. This event can have a filter definition in XML format as shown in Figure 3.2. We have created a filter definition, which encompasses a normal attribute filter, a *'MultipleSelectionList'* on an attribute and the *'Range'* operator. The path of the directory in which this XML file is stored and is known to the POM through the configuration file.

```
<EVENTNAME>
SpecialTicket
</EVENTNAME>

<MULTIPLESELECTIONLIST>
<ATTRNAME> FromCity </ATTRNAME>
<TYPE> String </TYPE>
<LISTVALUES> Gainesville, Jacksonville, Orlando, Tampa
</LISTVALUES>
</MULTIPLESELECTIONLIST>

<ATTRFILTER>
<ATTRNAME> ToCity </ATTRNAME>
<TYPE> String </TYPE>
<OPERATOR> Equal </OPERATOR>
</ATTRFILTER>


<ATTRFILTER>
<ATTRNAME> Price </ATTRNAME>
<TYPE> Integer</TYPE>
<OPERATOR> Range </OPERATOR>
</ATTRFILTER>
```

Figure 3.2 Filter Definition for SpecialTicket event

A subscriber would subscribe to the SpecialTicket event by specifying values for the various attributes defined in the filter in Figure 3.2. Below, we will look at an example scenario with two subscribers.

Subscriber 1	Subscriber 2
Name = "Subscriber A"	Name = "Subscriber B"
subIPAddr = "128.227.176.42"	subIPAddr = "128.227.156.67"
Email = ashenoy@cise.ufl.edu	Email = ahendro@cise.ufl.edu
Event = SpecialTicket	Event = SpecialTicket
FromCity = 'Gainesville', 'Jacksonville'	FromCity = 'Orlando', 'Jacksonville'
ToCity = 'Boston'	ToCity = 'Boston'
Price_Min = 250	Price_Min = 300
Price_Max = 400	Price_Max = 300

Figure 3.3 Example subscribers

3.2.1 Store Subscriber Profile

When the Event Manager receives the subscription information, it calls the POM '*storeSubscriberProfile*' API and sends the subscriber's as a parameter to this API. The POM first checks if a relational table exists for the given event name. If not, it uses the XML-Relational parser to create one or more tables. In this example scenario, we have one multivalued attribute. Hence, the two tables in Figure 3.4 will be created.

The filterId in the **MultipleFromCitySpecialTicket** table references the filterId in the **SpecialTicket** table. The subscriber profiles are then stored in the **Subscriber** table, the **SpecialTicket** table and the **MultipleFromCitySpecialTicket** table. Figure 3.5 shows the set of '*INSERT*' statements generated by the POM to store the subscriber profile into the persistent store.

SpecialTicket	
filterId	INTEGER
ToCity	VARCHAR(30)
Price_Min	INTEGER
Price_Max	INTEGER

MultipleFromCitySpecialTicket	
filterId	INTEGER
FromCity	VARCHAR(30)

Figure 3.4 Database Tables

Insert into **SpecialTicket** (filterId, ToCity, Price_Min, Price_Max) values (1,'Boston','250','400')

Insert into **MultipleFromCitySpecialTicket** (filterId, FromCity) values (1,'Gainesville')

Insert into **MultipleFromCitySpecialTicket** (filterId, FromCity) values (1,'Jacksonville')

Insert into **Subscriber** values ('Subscriber A', 128.227.176.42', 'ashenoy@cise.ufl.edu', 'SpecialTicket', 1)

Insert into **SpecialTicket** (filterId, ToCity, Price_Min, Price_Max) values (2,'Boston','300','300')

Insert into **MultipleFromCitySpecialTicket** (filterId, FromCity) values (2,'Orlando')

Insert into **MultipleFromCitySpecialTicket** (filterId, FromCity) values (2,'Jacksonville')

Insert into **Subscriber** values ('Subscriber B', 128.227.156.67', 'ahendro@cise.ufl.edu', 'SpecialTicket', 1)

Figure 3.5 Example Insert Statements

3.2.2 Filtering Mechanism

When an event is posted, all the subscribers of that event whose notification conditions are satisfied are notified of the occurrence of that event. In our sample

scenario, we will assume the occurrence of a SpecialTicket event with the following values

- FromCity – Jacksonville
- ToCity – Boston
- Price – 350

In this case, the Event Manager would call the POM '*getSubscribersOfEvent*' API with the SpecialTicket event object as a parameter, to perform the filtering of subscriber profiles. The '*getSubscribersOfEvent*' API will generate the following query to perform the filtering process.

```
Select distinct SubscriberName, IPaddress, SubEmail
From Subscriber a, SpecialTicket b, MultipleFromCitySpecialTicket
where a.eventName = 'SpecialTicket' and a.filterId = b.filterId
and b.filterId = MultipleFromCitySpecialTicket.filterId
and (FromCity= 'Jacksonville') and b.ToCity = 'Boston'
and b.Price_MIN <= 350 and b.Price_MAX >= 350
```

Figure 3.6 Example Query

The result of executing the query will be as shown below:

Table 3.1 Query Result

SubscriberName	IPAddress	SubEmail
Subscriber A	128.227.176.42	ashenoy@cise.ufl.edu

3.2.3 Performance

The POM's filtering mechanism is found to be considerably faster than the previous filtering mechanism used in the Event Server.

3.2.4 Other APIs

The implementation details of the other APIs have been given in Chapter 2. The usage of those APIs is straightforward. Hence, we have not provided example for using those APIs.

CHAPTER 4
SUMMARY AND CONCLUSIONS

Interfacing object applications to the relational data stores poses a major problem when building object-oriented applications. This area is still in the formative stage and is therefore technically challenging. The issue of translation between OO representation and relational model remains problematic. While solutions that can automate the translation process do exist, most of them are not adequate for complex objects such as the abundant object class libraries provided by Java. This motivated us to create an Object-Relational mapping framework, by devising on our own mapping standards. We have called it the Persistent Object Manager (POM) for Java applications.

The main features of POM are outlined below:

(1) POM automates the code generation for the object-relational mapping process and is transparent to the object-oriented application programmer.

(2) POM has formulated a set of object-relational mapping strategies, which take into consideration object-oriented concepts like Inheritance and Aggregation.

(3) POM provides APIs for performing object storage, retrieval, updation and deletion for general-purpose applications written in Java. It also provides APIs customized to provide persistence to the Event Server application.

(4) POM has been implemented using an Object-Relational database called Cloudscape, which is written in Java. Therefore, the database is platform-independent.

(5) POM utilizes the underlying DBMS facilities to provide strong support for persistent data storage, management, backup, transaction, concurrency control, security, etc.

In Chapter 1, we gave an introduction to **persistence** and the motivation behind developing POM. Chapter 2 provided the design and implementation details of the entire system where we looked at the design and implementation of the general-purpose module and also the customization for the Event Server application. In Chapter 3 we demonstrated the use of the POM by using some sample scenarios.

In conclusion, we would like to bring to the attention of readers that universal standards and guidelines that adequately address Object-Relational translation are still absent. Stabilizing these standards is a field for future research and will continue to evoke added attentions.

LIST OF REFERENCES

[1] Booch, C. "Object-Oriented Analysis and Design with Application," second edition, The Benjamin/Cummings Publishing Company, Redwood city, CA, 1994.

[2] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. "Object-Oriented Modeling and Design," Prentice Hall, Englewood Cliffs, NJ, 1991.

[3] Stroustrup B. "The C++ Programming Language," second edition, Addison-Wesley Publishing Company, Reading, MA, 1993.

[4] Sun Microsystems, Inc. "Java™ Object Serialization Specification," November 1998 http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/serial-title.doc.html

[5] Loomis, M. "Object Database: The Essentials," Addison-Wesley, Reading, MA, 1995.

[6] Grueva, M. "A Framework for Event Registration, Filtering, and Notification over Internet," Master's Thesis, University of Florida, August 1999.

[7] Informix Corporation. "Cloudscape Reference Manual," December 2000 http://www.cloudscape.com/docs/doc_36/doc/html/coredocs/rmtit.htm

[8] Informix Corporation. "SQL-J Language Reference," December 2000 http://www.cloudscape.com/docs/doc_36/doc/html/coredocs/sqljtoc.htm

[9] Informix Corporation. "Learning Cloudscape: The Tutorial," December 2000 http://www.cloudscape.com/docs/doc_36/doc/html/tutorial/tuttit.htm

[10] Melton, Jim, & Alan, Simon R. "Understanding the New SQL: A Complete Guide," Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[11] Gupta, S. and Scumniotales, J. “Integrating Object and Relational Technologies,” 2000 http://www.vbonline.com/vbonline/vigor/mapper/paper.htm

[12] Brown Kyle, & Whitenack. Bruce, “Crossing Chasms: A Pattern Language for Object-RDBMS Integration-The Static Patterns,” Technical Report, 2000, http://www.ksc.com/article5.htm

[13] Keller, Wolfgang, “Mapping Objects to Tables – A Pattern Language,” EuroPLop 1997 http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/mapping04.pdf

[14] Campione, Mary, & Walrath, Kathy, “The Java Tutorial: Object-Oriented Programming for the Internet,” (Second Edition), Amazon Inc., 1999. http://java.sun.com/docs/books/tutorial/

[15] Sun Microsystems, Inc. “Java™ Core Reflection API and Specification,” February 4th 1997 http://java.sun.com/products//jdk/1.1/docs/guide/reflection/index.html

BIOGRAPHICAL SKETCH

Anuradha Shenoy was born on March 26, 1975, in Mangalore, India. She received her Bachelor of Engineering degree from Karnatak University, India, in August 1997, majoring in computer science and engineering. She joined Tata Unisys Ltd., India in September 1997. She worked as a senior software engineer until December 1998.

She joined the University of Florida in August 1999 to pursue a master's degree in computer and information science and engineering. She worked as a graduate assistant during her master's studies at the Medical Informatics Center, Department of Radiology, in Shands Hospital.

Her research interests include object-relational databases and database internals.