University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations

2002

Layered approach to persistency modeling in object-oriented environment.

Kamran. Choudhery
University of Windsor

Follow this and additional works at: http://scholar.uwindsor.ca/etd

Recommended Citation
Choudhery, Kamran., "Layered approach to persistency modeling in object-oriented environment." (2002). *Electronic Theses and Dissertations.* Paper 2583.

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Layered Approach To Persistency Modeling In Object Oriented Environment

By

Kamran Choudhery

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the University of Windsor

Windsor, Ontario, Canada
2002

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-75821-4

Canada

972589

© 2002 Kamran Choudhery

Abstract

Object persistence is a fundamental feature of any object-oriented environment. Several programming language specifications do not include or discuss any method of providing persistence for objects. Several schemes have been developed for adding persistence in several different programming languages. Some of them require persistent objects to be allocated and treated differently than non-persistent objects, while some others require the programmer to provide vital parts of the persistence mechanism. It is desirable to make the persistence feature transparent, but the nature of object oriented programming makes it difficult.

This thesis proposes an approach to build a persistency layer in an object oriented programming language based, in part, on work by Ambler.

Dedication

To my Mom whom I love and admire!

Acknowledgments

First, I want to thanks GOD for all his bounties, favors and blessing upon me. This thesis would never have been completed without his mercy.

Words fail to express my indebtedness to Dr. R. D. Kent for his guidance and enthusiasm and who made this thesis a very enjoyable endeavour for me, I am grateful to my advisor for his guidance and advice during the development of the research presented in this thesis as well as for his generous financial support. I have been fortunate to have Dr. Kent as an advisor and to have the honor of working with him. I am also dedicating this thesis to him.

I would also like to thank Dr. A. K. Aggarwal from department of Computer Science for his valuable statistical guidance, support, impeccable understanding and for his friendly encouragement that allowed me to write this thesis.

I would also like to thank Dr. R. S. Lashkari from department of Industrial and Manufacturing System Engineering for generously taking time to read an earlier version of this thesis Proposal and to offering many insightful comments and suggestions for improvement.

I would also like to thank Dr. Alioune Ngom from department of Computer Science for his kindness and agreeing to join my thesis committee as a chairman.

I am deeply indebted to Dr. Mark Perry from the University of Western Ontario (school of computer science) for suggestions and encouragement on this thesis and to offer me to join his research team from coming September.

Three people have always provided unconditional love and support: my mom, my elder brother Imran Choudhery and Arslan Khan. Instead of pressuring me to achieve goals,

Mom and Brother motivated me to extend myself by saying, "As long as you do your best, we'll be proud of you". They always encouraged me to pursue what I enjoyed. They gave me the determination (often mistaken for stubbornness) and morals to accomplish objectives the honest (but often most difficult) way and to fight for the rights of others and myself. They always provided me with the best of everything. Despite our aggressive interactions as kids, my brother blossomed into a considerate and supportive sibling. My Mom has waited so long for this moment to come true; I am glad that their waiting has finally been rewarded.

For all I know, however, this thesis would have never come to light without Arslan's generous support and his kindness in allowing me to serve as dramaturg in his production. I am thankful to Mr. Khan for his friendship, advise and support throughout my graduate work. I am also dedicating this thesis to him.

I am also grateful to the Dr.Ahmed Tawfik and Dr.Walid Saba who provided discussions, ideas and suggestions essential to the planning and execution of this research.

I also wish to acknowledge the SHARCnet research fellowship program for funding received during summer of 2002.

Last but not least, I would like to dedicate this thesis to all my Friends, Arslan Khan, Arshad Shaikh, Rabhie Neouchi, Nabeel Abdullah, Adlane habeed, Aniss Zakaria, Amit Anand, Neeta Majmudar, Nomi, Vineet, Faris, Raheel Ahmed and special thanks to Lubna Batool. Indeed, it was my need to live up to their expectations of me that encouraged to put as much of myself into this work as humanly possible.

Table of Contents

Abstract		iv
Dedication		v
Acknowledgments		vi
List of Figures		xiii
Chapter 1		
	Introduction	1
	1.1 Thesis Statement	3
	1.2 Thesis Structure	5
Chapter 2	Background	7
	2.1 Historical Background	7
Chapter 3	Persistence	12
	3.1 What is Persistence	12
	3.2 Object's Lifetime	14
	3.3 Persistence as Extending an Object's Lifetime	15
	3.4 Forms Of Persistence Layer	17
	3.4.1 File Based Persistence	18
	3.4.2 RDBMS Based Persistence	18

Table of Contents

	3.4.3 ODBMS Based Persistence	19
	3.5 Kinds of Persistence Layers	21
	3.6 The Class-Type Architecture	24
Chapter 4	Durable Persistence Layer	28
	4.1 Durable Persistence Layer	28
	4.2 Why Eiffel	29
	4.3 Functionality for Durable Persistency Layer	33
	4.3.1 Feature Overview	33
	4.3.2 Main Characteristic of a DPL	34
	4.4 Example Place where DPL can be used	35
	4.5 Storing an Object	36
	4.6 Deleting An Object	37
	4.7 Retrieving Objects	37
	4.8 Full encapsulation of the persistence mechanism(s)	38
	4.9 Multi-object actions	38
	4.10 Transactions	38
	4.11 Extensibility	39
	4.12 Object Identities	39
	4.13 Conceptual Structure Of DPL	41

Table of Contents

	4.14 Security	42
	4.15 Application Security	42
	4.16 Tracking Operations	42
	4.17 Consuming Unreasonable Amounts of Resources	43
	4.18 Confidentiality of Authentication Data and Persistent Data	44
	4.19 Network Security	45
	4.20 Sending sensitive Data Over the network	45
	4.21 Concurrent Use	46
	4.22 Logging	47
Chapter 5	The Persistence Manager Pattern	49
	5.1 Context	49
	5.2 Problem	49
	5.3 Motivation	49
	5.4 Solution	50
	5.5 Overview of the functionality of the Persistence Manager	51
	5.6 Resulting Context	52
	5.7 Rationale	52

Table of Contents

Chapter 6	Implementation Of DPL	55
	6.1 Storage independent Steps	56
	6.1.1 Inherit from persistence and provide the features needed	56
	6.1.2 Provide the factory	59
	6.1.3 Provide the ondemand	59
	6.1.4 registering the persistent classes with the PM	60
	6.2 Usage Of DPL	62
	6.3 When an object needs to be saved	62
	6.3.1 Storing objects	62
	6.3.2 Retrieving objects	63
	6.4 Summary	63
Chapter 7	Testing and Post-analysis	64
	7.1 DPL Compared to Embedding SQL Directly	64
	7.1.1 Sample Code	64
	7.2 Sample Code	66
	7.3 Tradeoffs	67
	7.4 Benchmark Information	68
	7.5 Benchmarks	69
	7.6 DPL Compared to Similar Components	71

Table of Contents

7.6.1 Object Relational Bridge 71

7.6.2 Advantages of OJB Compared to DPL 72

7.6.3 Disadvantages of OJB Compared to DPL 72

7.7 Benchmarks 74

Chapter 8 Conclusion 77

8.1 Conclusion 77

8.2 Future Development 78

References 82

Glossary 87

Appendix A Sample Persistence Classes 92

Vita Auctoris 103

List of Figures

Figure 3.2	Object Persistence	14
Figure 3.5	Hard-coding SQL in your domain/business classes	21
Figure 3.5.1	Creating data classes corresponding to domain/business classes	22
Figure 3.5.2	Durable persistence layer	23
Figure 3.6	The class-type architecture	25
Figure 4.1	Architecture Of Sample application Using DPL	35
Figure 4.13	Conceptual Structure of DPL	41

Chapter 1

Introduction

Object-oriented programming (OOP), like most newer paradigms incorporates and, builds on old ideas, extending them, and permitting novel interpretations and approaches. A revolutionary concept that changed the rules in computer program development, object-oriented programming is organized around objects rather than actions, data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

The programming challenge once revolved around how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate rather than the logic required to manipulate them.

A basic task for Object Oriented software is producing, changing and viewing persistent objects. Persistent objects are objects that exist beyond the lifetime of the application; this is typically achieved by storing them in some kind of a data store, such as a file or, most commonly a relational database (see section 3.4). There are several recurring problems that application programmers have to deal with when developing object-oriented applications that use relational databases as a persistence mechanism. The greatest problem is handling changes in the data structure - a simple alteration forces some reworking in both the application persistence logic and the database. Larger changes - e.g. switching the underlying database from one vendor to another - can require a great amount of work.

Another concern is application design – the persistence logic of many applications is similar, differing mainly in the exact data structure. It would be favorable to exploit this similarity and produce an automated solution.

My thesis proposes a reusable generic approach to handling persistence logic namely, the Persistence Manager design pattern, which solves several problems when using relational database management systems in an application environment. This pattern represents a solution where business logic is separated from persistence logic to the degree where persistent objects are not aware that they are being stored and retrieved transparently. The solution is easily customizable for the data structure of different applications. It provides fully automated persistence - the application programmer has no need for writing any database access code, the component handles all data storage, retrieval and deletion by itself. We propose the Persistence Manager to be a well-designed approach to persistence logic, and to be a viable software component.

The proposed persistency layer encapsulates the entire persistence logic of an application, and provides access to persistence via simple operations like store (object), delete (object) and retrieve (criteria). In addition to solving the problems listed above, using the Persistence Manager approach yielded better modularized application structure, faster program development (as the programmer does not need to develop any database access code) and possibility for better performance on object retrieval.

1.1 Thesis Statement

This thesis offers a solution to these concerns.

A. Implementation of Durable Persistence Layer

B. Persistence Manager Design Pattern

C. Comparison of DPL

First, we present a component that we wrote as an implementation of such software-the object persistence layer. Based on a variety of different terminologies used in literature we have adopted the name durable persistence layer (DPL), because of its robustness and durability characteristics. Our DPL, written in the Eiffel programming language, provides application objects with transparent persistence. With transparent persistence, persistence of objects is provided automatically and the logic for performing persistence operations is expressed in the other languages, without the client programmer needing to know anything about databases and Structured Query Language (SQL). Persistence Layer encapsulates all the persistence logic that an application needs, and gives access to persistence via simple operations like *store(object)*, *delete(object)*, and *retrieve(criteria)*.The current implementation embodies ideas from Scott Ambler's white paper " *The Design of a Robust Persistence Layer for Relational Databases*." [Amb00b].

Second, we introduce the Persistence Manager design pattern[1] for a reusable persistence layer that separates application logic from persistence logic. The basic idea for such software was introduce in Scott Ambler's white paper: "*Mapping Objects To Relational Databases*" [Amb00a].

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

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

1.2 Thesis Structure

In the remainder of this chapter we present the structure of this Thesis.

Chapter 2 mainly discusses the historical background of persistence programming systems. The chapter also states some known limitations, lists related work and analyzes the problem of changes in data structures.

Chapter 3 introduces some prominent issues in persistence mechanism. The chapter focuses on Persistency layer, forms of persistence, kinds of persistency layers and requirements of persistency layers.

Chapter 4 establishes the motivation for our work. In this particular section we present a component that is written as an implementation of the object persistence layer software. We christened it as DPL (durable persistence layer) because of its robustness and durability characteristics. We also discuss the reason of choosing Eiffel in this chapter. It also focuses on how DPL can be used in the architecture of a sample application. The chapter also emphasizes the security issues concerning DPL and its possible solution.

Chapter 5 introduces the Persistence Manager design pattern for a reusable persistence layer that separates application logic from persistence logic.

Some fundamental ideas for implementing DPL are presented in Chapter 6. Comparing the benefits of using the *Persistence Manager* pattern with the basic approach to persistence logic - embedding database access directly into the application code - and judge the usefulness of the pattern are presented in Chapter 7. This Chapter also presents comparison of DPL with similar publicly available Java software that at least in some terms follow the pattern and offer similar functionality, and the viability of DPL.

Finally, we proceed in Chapter 8, to draw conclusions and then offering suggestions of possible approaches, which could be investigated in the future development. In addition we include a glossary of terms for ease of reference and also an Appendix in which we list the Classes used in our software implementation.

Chapter 2
Background

2.1 Historical Background

Research in persistent programming systems started in the late 70's, when it was noticed that storing 'long-term' data in a different logical framework from 'short-term' data leads to all sorts of problems in large and complex applications. An analysis by IBM showed that around 30% of the code of long-lived, large scale applications was devoted to the movement of data in and out of the programming language domain. The fact that this code is notoriously susceptible to system evolution errors, coupled with the statistic that 2% of the USA's GNP is spent on software 'maintenance', leads us to believe that storing long-term data in a file or database system is expensive.

A solution may be provided by *persistent programming languages. A* somewhat misnomer 'persistent programming languages' is shorthand for 'programming languages where persistence is treated as an orthogonal property of data', rather than programming languages, which survive for a long time [ADK01]. In such languages the treatment of data is entirely independent of its longevity - no matter for how long the data persists, the data model is unaffected.

" The reason programming languages treat persistent data differently from transient data is largely historical, based on the different physical properties of RAM and other storage devices. Although modern virtual memory makes a mockery of this distinction,

only a few programming languages can claim an orthogonal treatment of persistence. Commercial vendors are now becoming increasingly aware of the importance of persistence as an aid to increasing the cost-effectiveness of the software life cycle [BDM00a]."

The merging of distributed computing and object- oriented technology has resulted in novel ways to design and develop modern, aggregate architectures of databases. When you first begin to design any application you face numerous issues, including how your application is going to communicate with your databases. For years, many architects and system developers have not given this issue of communications interface design the attention demands.

The most common approach is to write code that talks directly to the database (Application Programming Interface) API provided in your environment. In the early days of SQL databases, your only approach was to use the database API provided by the database vendor, such as Oracle's OCI (Oracle Call Interface). This solved the initial need to enable your application for a particular database. However, your application was then tied to that database. You could not easily support another database, if at all, due to its different database-specific APIs, as well as the prohibitive cost of parallel development for each API.

Eventually, standard database APIs were introduced which worked for any database. This effort began with the development of ODBC (Open Database Connectivity). ODBC was

the first standard API to address the problem of database-specific APIs. ODBC allows you to issue SQL calls directly to the database. This API is function based, not based on objects. In order to talk to ORACLE, you need to install the ORACLE ODBC driver, but your application has no specific knowledge regarding the driver. The driver simply implements a standard interface. This allows you to develop applications that can talk to many different databases, providing that you use standard SQL (and the database has an ODBC driver). Once you begin to use the specific features of a particular database, however, your application becomes dependent on that database and supporting other databases becomes an issue.

If you are passing SQL statements from your code through your database API (such as JDBC), your application will become tightly integrated with your database schema. This is commonly referred to as injecting your schema into your application code. Once the schema is injected, it becomes difficult to modify or enhance your schema without modifying your code. At this point, you must find all the places in your code that are impacted by the potential change before you can update the code. This problem usually presents itself after the first version of your product, forcing you to modify your schema based on a new requirement or a particular user situation. The maintenance cost of this can be huge [BDM00b].

Later, Microsoft provided OLE DB, followed by ADO. Each simplified the API by providing objects for creating connections and statements, but the issue of SQL residing in your application code remained.

The corresponding common database API in the Java domain is JDBC (Java Database Connectivity) and it raises similar issues to those of ODBC. So what did developers do? Many developers created a library of SQL statements in their code and provided a public API for the rest of the application developers to call. This encapsulated all the database calls into a central location that could be maintained and updated appropriately as the application matured. This does not solve the problem of having to update the code when the database schema changes, but it makes it much easier to manage.

After you have developed several applications or systems following this approach, you will begin to see the problem it creates. As your application matures and additional features are added over time, your application accesses the same database table from several locations in your code. The application modifies the same table in several different places in your application, sometimes in different ways. This can make it very difficult to track down and isolate problems in your code.

Most applications today start with a simple prototype. When developing a prototype, it is usually much faster to just write the SQL statements and put more focus on the UI and the user tasks (which are important). The next important step would be to add the building blocks to your application that are necessary to give it a reasonable chance of success in supporting new requirements. Unfortunately, this is not always done.

Instead of writing code in terms of database connections and SQL statements (which was the standard way of thinking about database access), developers started writing classes that

presented the data in the form of Persistent Objects. Persistence refers to something (or a property) that exists beyond its lifetime. As applied to an object-oriented programming language, persistence describes objects that exist for an extended period of time, often beyond the lifetime of the original program that created the objects. There are several forms of persistence available to programmers (see section 3.4). The forms include file-based persistence, relational databases, and object databases; but the most common persistence mechanism at the present time is a relational database management system. Relational databases are an efficient and proven technology, they have widespread support in development languages and third party tools, and practitioners are familiar with them. Several problems arise in combination of using an object-oriented language (like, Eiffel, Java, SmallTalk or C++) as the development environment and a relational database as the persistence mechanism, however. Another problem is the impact of changes in data structure. If the class structure changes in some way, then, besides the database, this triggers changes in the persistence logic of the application as well. For example, if a field type is changed from integer to floating-point value, then even this minor modification induces a change in the persistence logic.

We conclude this chapter with the following quote that provides emphasis of the need for effective persistence mechanism:

> "Persistence has proven the technical competence of its software offerings with a growing number of large and demanding customers, including Air Canada, AT&T, BellSouth, Boeing, Celera, Cisco, CNP Assurances, FedEx, Instinet, Intel, JP Morgan, Lucent, Morgan Stanley, Norwest, Pfizer, Qualcomm, Scientific Atlanta, and Solomon Smith Barney" Ed Murrer (Senior Vice President Persistence Software)."

Chapter 3
Persistence

3.1 What Is Persistence?

Persistence has been defined previously. Nonetheless, it is useful to begin this chapter with a listing of several distinct scenarios, each of which illustrate various aspects of persistence.

- One of the most critical tasks that applications have to perform is to save and restore data. Without it, software would be little more effective than the typewriter - users would have to re-type the data to make further modifications once the application exits.
- Persistence consists of data retrieval from one or more data stores into system memory after data manipulation (i.e. applying business rules) and storage of the modified data into the appropriate data store.
- After program execution the state of an object (the value of its attributes) is lost unless saved to permanent storage. When an object maintains its state between program executions it is said to be persistent.
- The issue of how to store an object in a permanent storage. Objects need to be persistent if they are to be available to you and/or to others the next time your application is run.
- Applied to an object-oriented programming language, persistence describes objects that exist for an extended period of time, often beyond the lifetime of the original program that created the objects. These include for example, executable images, translation Code in JVM byte code.

Persistence is a critical architectural mechanism found in most business systems today. Examples of emerging, advanced computing systems, or platforms, with significant need for persistence are computing grid system (e.g. Sharcnet, WestGrid, and many others)*.

*www.gridcanada.com

3.2 Object's Lifetime

An executing system represents application data as a collection of collaborating objects, connected by a potentially complex web of references. With current technology, these objects occupy volatile memory in the computer and vanish when the system terminates. Persistence extends the lifetime of an object from one execution of a system to the next.
It is achieved by saving the object to a file on secondary storage in one session and restoring it from the file in another. *Persistence mechanisms do not refer only to traditional disk storage; rather, they include also additional, new techniques for distributed memory mapping and emerging optical network storage technologies.*

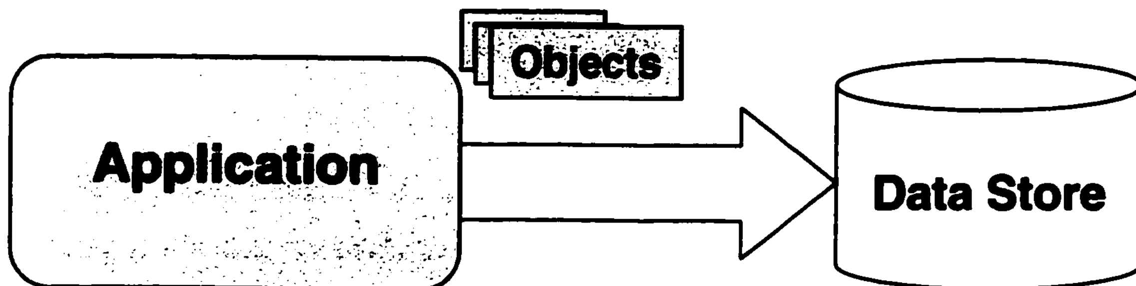


Figure 3.2 Object Persistence

Figure 3.2 present that Persistent objects are objects that exist beyond the lifetime of the application; this is typically achieved by storing them in some kind of a data store, which most commonly is a relational database.

New OO programmers learn that objects have a lifetime. An object begins its life when created by the **new** operator (for example, **new** String("hi")). After it is created, the object

exists until destroyed by the Java Virtual Machine's garbage collector; an object can be garbage collected only when the Java program no longer holds a reference to the object. Objects can also be destroyed implicitly, when the Java program ends. The following code snippet demonstrates the essential concepts of Java object lifetimes:

Example 3.2: A simple object example

```
{
  Date d = new Date();                  // Create Date object-d  starts its life
  System.out.println(d.toString());     // Date object still exists by virtue of referencing
}                                       // Date object is no longer referencable and may be destroyed.
```

In this example, a new `Date` is created within a program block (`{}`) and stored in a variable (`d`) local to that block. Upon reaching the ending curly brace (`}`), the local variable `d` exists no longer. From that moment, the `Date` object that was created is no longer referencable and may be garbage collected.

3.3 Persistence as Extending an Object's Lifetime

Persistence is a way to extend the lifetime of an object beyond the lifetime of the program that created it. To understand why it is useful to have persistent objects, consider an `AddressBook` class that contains names, addresses, and telephone numbers:

```
public class AddressBook {
  public String[] names = null;
  public String[] addresses = null;
  public String[] phonenums = null;
}
```

A person writes information in an address book so that it is available at a later date, when the information is needed. Most people are unlikely to remember addresses and telephone numbers, so they write that information into a book. If you try to use the `AddressBook` class to represent a real address book, you will find that it does not support the "save it now, use it later" paradigm. All instances of the `AddressBook` class are destroyed when the Java program ends.

To be useful, an `AddressBook` object must exist for an extended period of time. It must be *persistent* (possibly for years). Every time the user looks up, adds, or modifies address information, the `AddressBook` object is needed. Because the program that uses the `AddressBook` isn't always running, the `AddressBook` must be preserved during the time the program is not running.

Persistence is usually implemented by preserving the state (attributes) of an object between executions of the program. To preserve state, the object is converted to a sequence of bytes and stored on a form of long-term media (usually, a disk). When the object is needed again, it is restored from the long-term media; the restoration process creates a new Java object that is identical to the original. Although the restored object is not "the same object," its state and behavior are identical. The following example outlines an API for a helper class that might be used to provide save and restore capabilities for `AddressBook` objects:

```
class AddressBookHelper {
  public static void store(AddressBook book, File file) {...}
  public static AddressBook restore(File file) {...}
}
```

To save an `AddressBook` to a file, you must explicitly write a few lines of code to store the object. The code might look like the following:

```
File output = new File("address.book");// persistent media
AddressBookHelper.store(addrBook, output);
```

Restoring an `AddressBook` from a file would look similar:

```
File input = new File("address.book");  // persistent media
AddressBook addrBook = AddressBookHelper.restore(input);
```

3.4 Forms Of Persistence Layer

There are several forms of persistence available for programmer. The forms discussed in this thesis include file-based persistence, relational databases, and object databases. These forms of persistence differ in several categories, including: logical organization of an object state, the amount of work required of the application programmer to support persistence, concurrent access to the persistent object (from different processes), and support for transactional *commit* and *rollback* semantics [AMB00b].

3.4.1 File Based Persistence

Files are often used to store information between invocations of a program. Data stored in a file may be simple (a text file), or it may be complex (binary strings). In daily use of a computer, you often interact with objects that are stored in files (e.g. word processing documents, spreadsheets, network diagrams, and so on).

A file-based persistence mechanism requires the programmer to put a significant effort into achieving persistence. The programmer must choose an external representation of the object, and write the code that saves and restores the objects.

Usually, concurrency control and transactional semantics do not apply to file-based persistence. Storing objects in files is usually appropriate for single-user applications that follow the `File/Open...` and `File/Save` model.

3.4.2 RDBMS Based Persistence

Relational database management systems (RDBMS) can also store persistent objects, but the characteristics of a relational database are different from file-based persistence. A relational database is organized into tables, rows, and columns, rather than the unstructured sequence of bytes represented by a file. There are two major ways to store objects in a relational database. The first option is to interact with the database on its terms. The JDBC API provides interfaces that directly represent relational database structures. These structures can be used and manipulated by using conventional method. The other option is to write your own classes and "map" between the relational data structures and your

classes. This type of mapping is a well-understood problem for which many commercial solutions are available.

When using a relational database, unless you are using a tool to perform database-to-class mapping, you must write a large volume of code to interact with the database. Managing objects in the database requires you to write SQL statements (inserts, updates, deletes, and so on), which are forwarded to the database through the JDBC API.

Although using a relational database involves more work, there are a few benefits. Relational databases usually support concurrency control and transactional properties. Multiple users can access the database without interfering with other users changes, because the database uses locks to safeguard access. Additionally, almost all relational databases support ACID properties (*i.e. atomicity, concurrency, isolation, durability*). These properties protect the integrity of the data by assuring that blocks of work (referred to as transactions) either complete successfully or are rolled back without affecting other users.

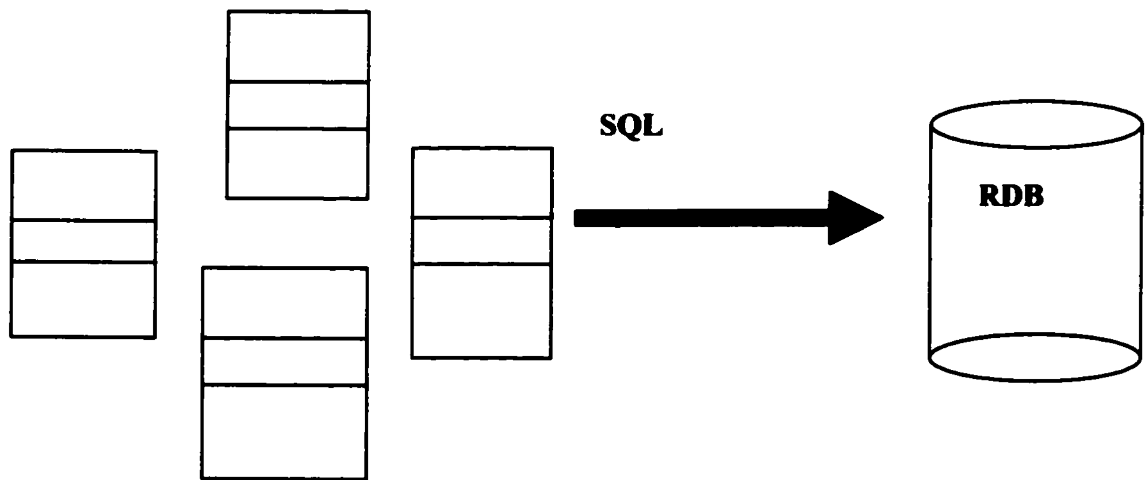
3.4.3 ODBMS Based Persistence

Object database management systems (ODBMS) support persistence in a different manner than file-based persistence and relational databases. The philosophy behind object databases is to make the programmer's job simpler. Object databases (as the name implies) store objects; the programmer does not have to write SQL statements or methods to package and unpackage objects; the object database interface usually takes care of those details.

Object databases usually support concurrency control and ACID properties, as with relational databases. They provide for concurrent access to the database, and they also provide commit and rollback transactional control.

3.5 Kinds of Persistence Layers

We would like to begin this section with a discussion of the common approaches to persistence that is currently in practice today. Figure 3.5 presents the most common, and least acceptable, 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 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 requires a reworking of your source code. Hard-coded SQL in your business classes results in code that is difficult to maintain and extend.



Domain Classes

Figure 3.5 Hard-coding SQL in your domain/business classes

Figure 3.5.1 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. 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.

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

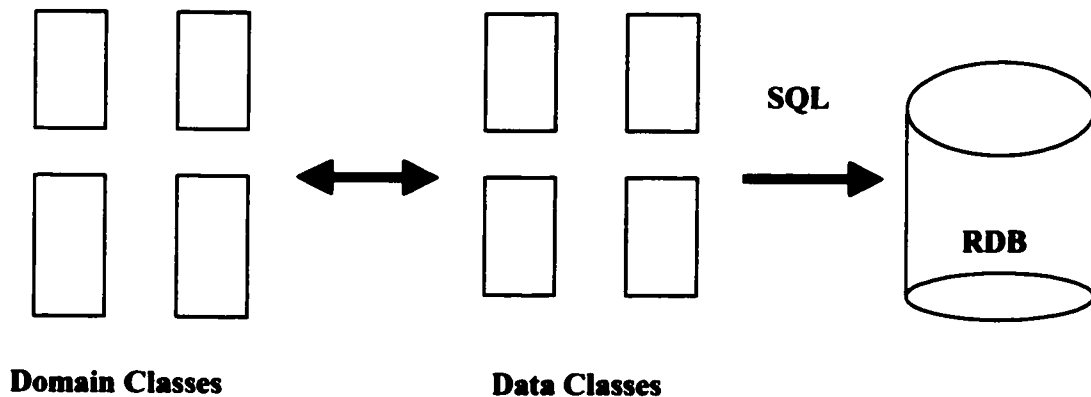


Figure 3.5.1 Creating data classes corresponding to domain/business classes

Figure 3.5.2 presents the approach that was adopted for our Durable 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 on your applications, a minor one if you build the layer well, but there is still an impact.

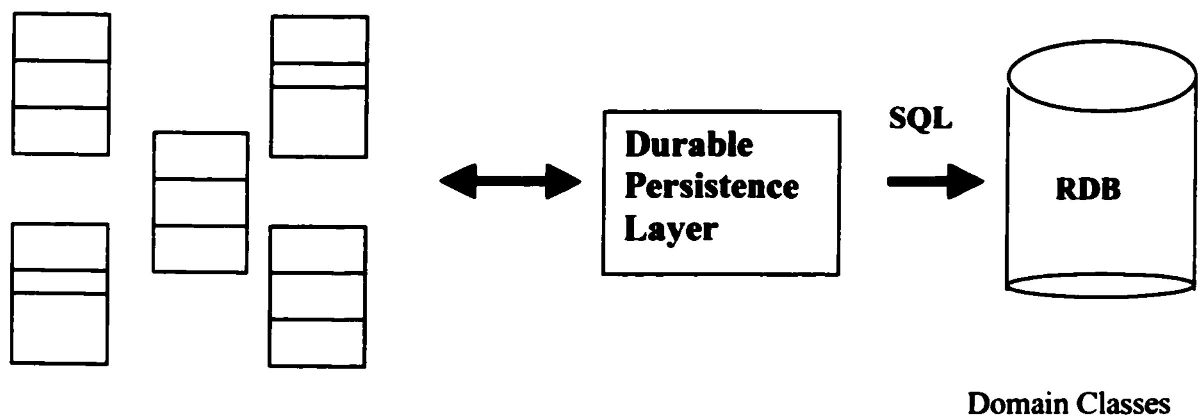


Figure 3.5.2 Durable persistence layer

3.6 The Class-Type Architecture

Figure 3.6 shows a class-type architecture that your programmers should follow when coding their applications. The class-type architecture is based on the Layer pattern proposal by Buschmann Etal [BFM96]. The basic idea is 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 one makes it easier to maintain and enhance because the coupling within an application is greatly reduced.

Layering your application code increases its robustness

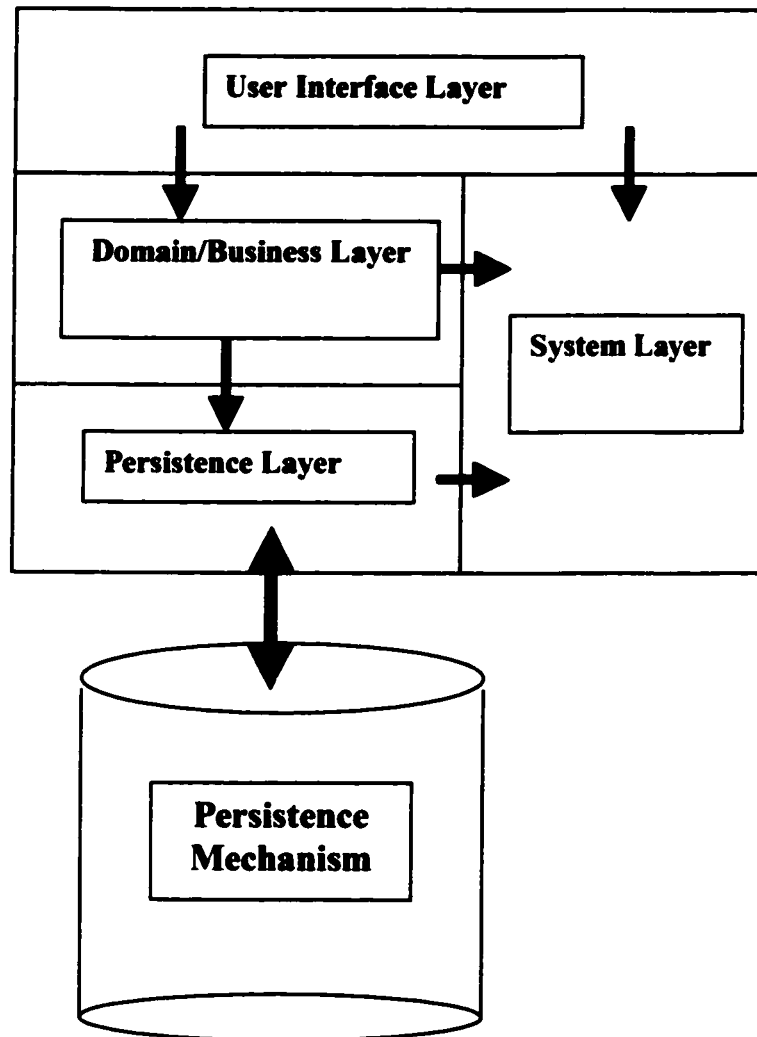


Figure 3.6 The class-type architecture

Figure 3.6 shows that users of an 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 intended to send messages to classes within the domain/business 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 **Phone Call**, and 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 3.6 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, 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.

Chapter 4
Durable Persistency Layer

4.1 Durable Persistency Layer

The persistence layer can be looked at as a way to access objects within the database. The Durable Persistence Layer (DPL*) is software that implements the Persistence Manager pattern. It is a component that provides access to persistence. The DPL pattern allows the application programmer to work with objects, which have the persistent property, without knowing anything about database query languages. The programmer only has to call methods for loading, storing and deleting of objects, the implementation of these methods is left to the DPL. In order to map objects and their relations to other objects (inheritance, association, aggregation...) to tables of a database, an object-relational-mapping is required that supplies patterns (e.g. how to map inheritance hierarchies to tables) with all the consequences.

** It is a common practice to give software products names that often are unrelated to the product domain. For example, Ant is a popular Java compiling tool, and Tomcat is a Java web component container. I propose the name "DPL" because of its durable and robust characteristics.*

Our motivation for using Eiffel as our development system is drawn from the properties that Eiffel is a pure object-oriented programming language designed with the explicit intent to produce high quality, reliable software. It is pure in its nature in that every entity is an object declared of an explicitly stated class type. It adheres to some of the long proven and time-tested programming practices that have made languages like Modula-2 a successful engineering advancement. Eiffel* promises to be the next step toward a better understanding and more importantly, efficient use of the object-oriented practices that have evolved thus far.

4.2 Why Eiffel ?

Eiffel* is a *methodology*, *language* and *environment* for developing high-quality reusable software components.

- *Methodology:* A system of principles and rules based on theory and practice that guides the software construction process to achieve high-quality software.
- *Language*: An elegant Object-Oriented Programming Language (OOPL) that strongly supports the methodology of this thesis.
- *Environment*: Integrated set of OO CASE tools that strongly support software development using the methodology and language.

*www.eiffel.com

Eiffel is an elegant modern object-oriented approach to software development that emphasizes *software product quality* - especially correctness and reusability - crucial prerequisites to shifting current practice towards component-based development. Some of Eiffel's goals can be realized in non-Eiffel approaches, but only with considerable external/non-standard mechanisms. Eiffel is elegant in the way it seamlessly integrates many aspects of high-quality software development. It is a proven and portable technology that strongly supports the requirements of large-scale, team-oriented development in many application areas, from embedded systems to distributed enterprise applications.

The Eiffel language includes many important features such as:

- Pure OO programming.
- Simple, highly readable syntax.
- Clean Multiple Inheritance.
- Static, strong-typing.
- Genericity.
- Automatic memory management (garbage collection).
- Disciplined exception handling.
- Persistence (low-level object database).
- Concurrency (threads and later SCOOP).
- Full support for Design By Contract.
- Interoperability with other languages such as C and C++.
- Large collections of high-quality reusable class libraries for many application areas.
- Quality development environment/tools.

- Mature, supported language. (NICE: Non-profit International Consortium for Eiffel*).

In another way Eiffel can serve as an excellent component combinator - wrapping and combining proven software components developed in languages other than Eiffel. This enables development teams to incorporate external code, such as Fortran libraries for numerical computation, into a flexible OO architecture. This is far better than trying to implement OO features in a non-OO language or even in another OO language that is less supportive of software quality than Eiffel.

Of course, the quality of the final result is limited by the quality of the wrapped component too - the "weakest link" notion - so it is important to ensure that components selected for wrapping be of verified high quality.

Eiffel involves a comprehensive discipline that ambitiously targets many key aspects of software development. It fosters, indeed requires, a shift in mindset that prioritizes certain qualities, such as correctness, robustness, understandability, testability, extendibility, reusability over others, such as functionality.

That is, the Quality First approach to development recognizes that some qualities simply cannot be retrofitted - they must be built-in from the beginning and upheld throughout the

*www.eiffel.com

entire development process while others, such as functionality, naturally increase over time. This is not a small step, but it is not a difficult one either. Eiffel goes far beyond paying lip service to good software engineering principles. It actively supports, and even enforces, these principle like no other development approach in widespread use. It does so effortlessly in a seamless integrated environment, unlike other approaches that require significant extra-linguistic mechanisms that try to inject and uphold quality. Eiffel's elegant simplicity makes it significantly easier to master than other OOPLs.

We conclude with the following two quotes to stress the qualities of Eiffel as a support tool for development.

> *"There are two things that [Eiffel] got right that nobody else got right anywhere else: support for design by contract, and multiple inheritance. Everyone should understand these "correct answers" if only to understand how to work around the limitations in other languages."* (*Paul Dubois in comp.lang.python Usenet newsgroup, 23 March 1997).*

> *"If it can be defined by just one sentence, Eiffel is a language to write Eiffel Libraries-that is to say to write the best possible, reusable industrial quality software components that we can think of." (Bertrand Meyer in .ExE Magazine).*

4.3 Functionality for Durable Persistency Layer

A DPL encapsulates the behavior needed to make objects persistent; in other words to read, write, and delete objects to/from permanent storage. DPL should support 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 object bases.

4.3.1 Feature Overview

Following are the few key features of DPL:

- uses relational databases for data storage, automating object-relational mapping
- stores objects in the data store
- deletes objects from the data store
- retrieves objects from the data store, via potentially complex retrieval criteria; can retrieve multiple objects with one operation
- supports one-to-many relationships between classes - when retrieving an object, its related objects can be retrieved automatically
- supports referential integrity - when storing an object, its related objects will be stored as well; when deleting an object, its related objects will be deleted with it
- caches persistent objects, which can yield a dramatic increase in retrieval speed
- can access multiple databases
- persistent classes do not have to inherit from a specific class

- supports composite identities
- custom SQL can be specified

4.3.2 Main Characteristic of a DPL

In this section we discuss DPL characteristic

Compatibility:

- Works with your existing frameworks and databases
- Works with your modeling tool(s)

Flexibility:

- Allow complex mappings and operations, but provides good defaults

Encapsulation of the persistence layer:

- Provides good (ideally complete) isolation from changes in the database(s)

Support for the features required by your model and data sources:

- Distributed transactions
- Required scalability and performance load balancing
- Concurrency and parallel query processing

4.4 Examples of cases where DPL can be used

Following are few examples of using DPL:

- web applications for information systems, which basically are database front ends and therefore rely heavily on stored data
- desktop applications, that need to store user preferences
- applets for informations systems, acting as database frontends
- any kind of application where there is a need of making object persistence

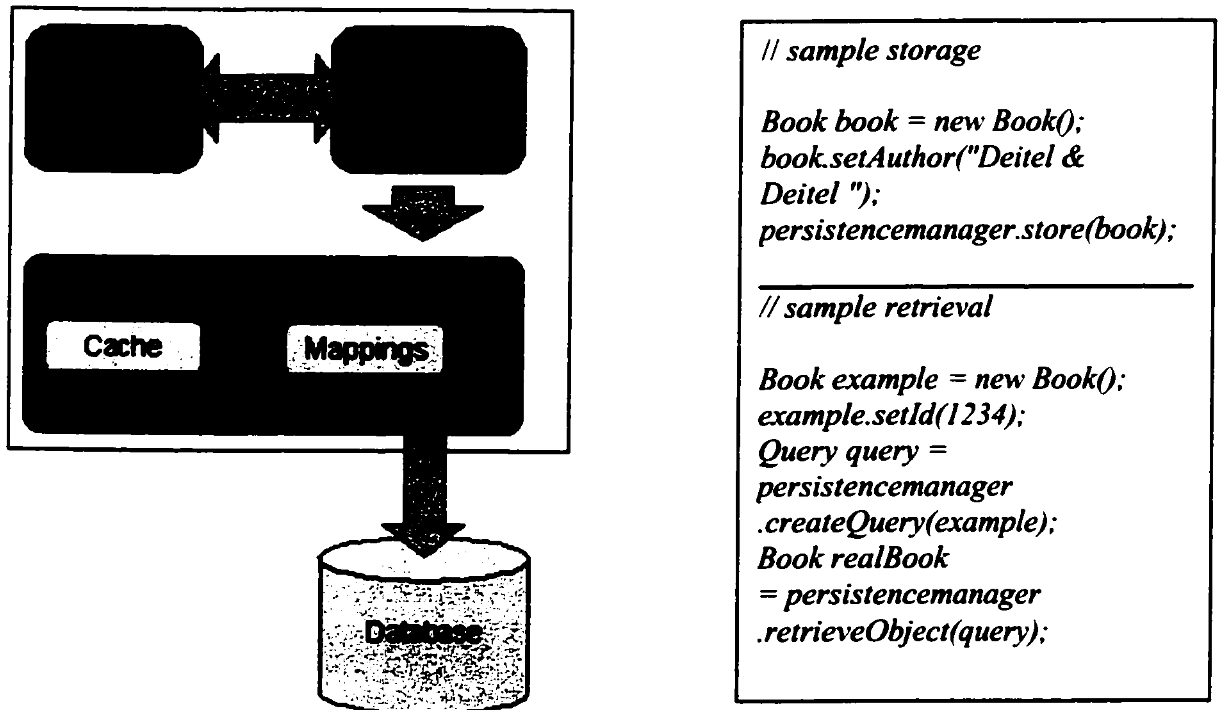


Figure 4.1 Architecture Of Sample application Using DPL

Figure 4.1 Display a simple application using DPL. DPL has been designed with regard to future support for other data stores besides relational databases. For example, if another package is provided that implements the Persistencemanager interface and uses XML files for data storage, then it can be integrated seamlessly with the existing code. Nothing changes for the client programmer -Persistence Managers of different type will be obtained in exactly the same way. This approach has the benefit of supporting the seamless integration of new types of Persistence Managers discussed above, as the configuration file specifies what kind of a Persistence Manager to use.

4.5 Storing an Object

Storing an object is very straightforward. The client programmer only needs to send a message to the Persistence Manager and the object gets stored. If the object's identity is unspecified, which is the case if the object has been transient so far; it is assigned a unique identity.

Example: 4.5 Storing an object:

```
// Create a new object and set some of its attributes
Book book = new Book();
book.setTitle("C How to Program");
book.setAuthor("Deitel & Deitel ");
// Store the object
broker.store(book);
```

4.6 Deleting An Object

To be able to delete an object from the underlying data store, the only information needes to know is its identity.

Example 4.6 Deleting an object:

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

4.7 Retrieving Objects

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

4.8 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. The persistence layer takes care of the rest of the transactional details. Furthermore, except for well-justified exceptions, you shouldn't have to write any special persistence code other than that of the persistence layer itself.

4.9 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.

4.10 Transactions

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.

4.11 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 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.

4.12 Object Identities

An object identity (OID) is an identifier that, assigned to a persistent object, uniquely identifies the object. In a relational database, tables have key columns. The values of the key columns of a row make up the identity of a row. If objects, mapped to relational databases, are to have identities, then these key columns (which usually have no business meaning) must intrude into the object classes (where they normally are not present). Classes are assigned attributes that map onto key columns in the database table. The intrusion is necessary for creating relationships between objects. If at run-time a persistent object has a reference to another persistent object, then when the objects are made persistent, this reference needs to be stored, and the only way to do it is via OIDs. For example, a master class Person has a detail class Address, and a Person object can contain several Address objects. When storing the Person object and its Address objects, the OID of the Person object is stored as a foreign key in the database table the Address class maps onto. When the Person object is materialized again, its Address objects are also materialized by using the Person's OID. Another use for OIDs in DPL is in caching - to cache an object, the object is mapped to its OID, and cache lookup is performed using the

OID. If the objects of a class have no identity, then they cannot be cached and the performance gain from caching is lost.

4.13 Conceptual Structure of DPL

Figure 4.13 represent a conceptual overview of the architecture of the Durable Persistence System and what basic parts it has and how they interact.

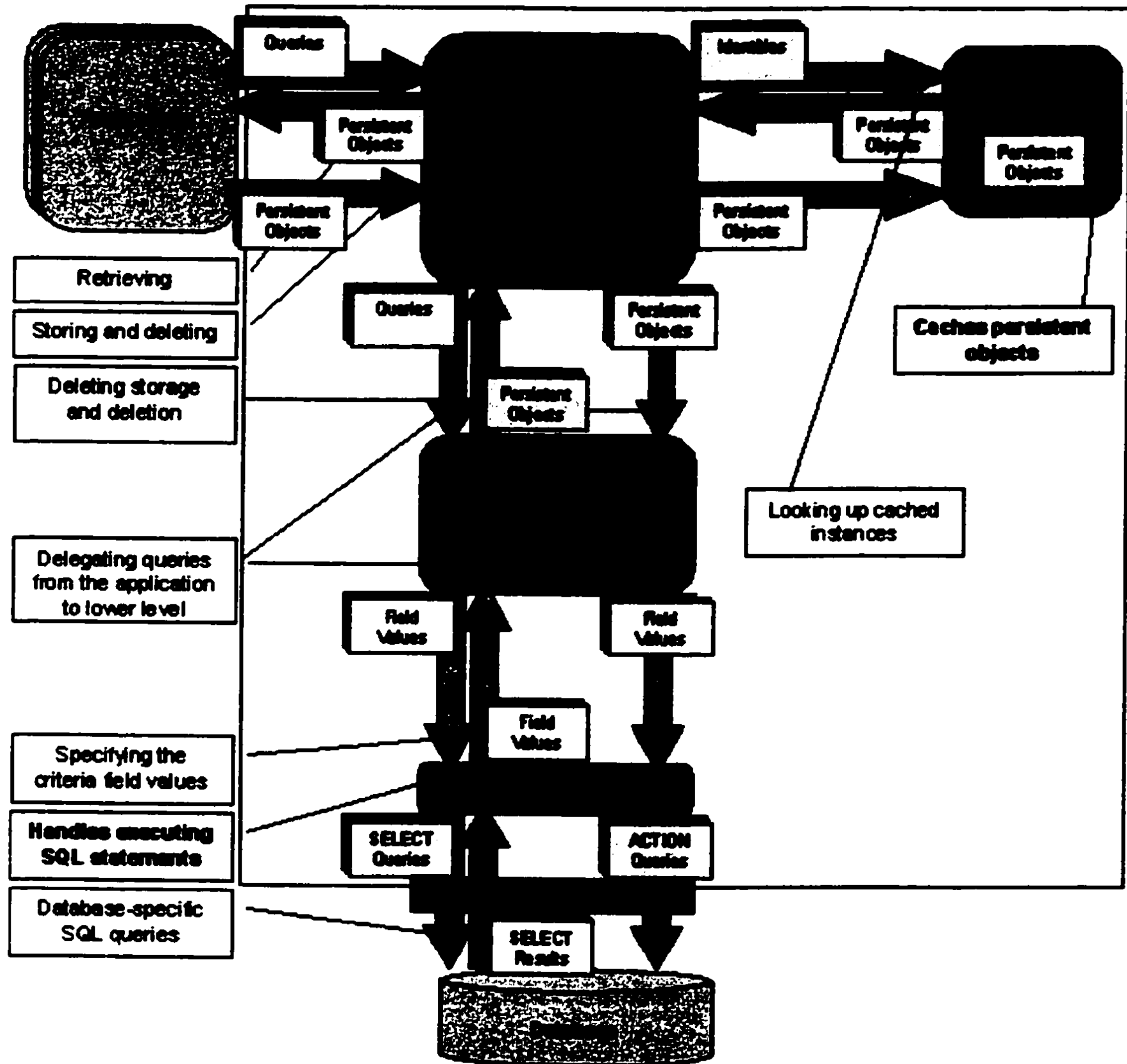


Figure 4.13 Conceptual Structure of DPL

4.14 Security

Following are the few security issues:

4.15 Application Security

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

In the following section 4.16 to 4.20, we present several distinct problem scenarios that illustrate some of the primary issues of concern. In each case we present the problem, an assessment and a solution strategy.

4.16 Tracking Operations

Problem

The history of potentially sensitive operations must be accessible.

Assessment

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

Solution

- Enable adequate logging in the persistence layer (See section 4.22 for a discussion of Logging).

4.17 Consuming Unreasonable Amounts of Resources

Problem

The persistence layer could use up too many system resources.

Assessment

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

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

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

4.18 Confidentiality of Authentication Data and Persistent Data

Problem

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

Assessment

- This concern usually arises if the application is a web application, for example a servlet, that by default is not run with the access rights of the user and thus the configuration file must have more liberal access rights. If no solution is used, then harmful activity on persistent data is at least detectable if adequate logging is enabled (See section 4.22 for a discussion of Logging).

Solution

Set the access permission of the file to be accessed by the attacked user only, and launch the application with the rights of the attacked user.

4.19 Network Security

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

4.20 Sending sensitive Data Over the network

Problem

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

Assessment

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

Solution

Use a secure ODBC driver. With a secure driver, the communication between the driver and the database is encrypted.

We note that this solution can only provide security up to a point of certified reliability that is limited by the qualities of the underlying algorithms and other security mechnasims.

4.21 Concurrent Use

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

PersistenceManagerFactory retains the Persistence Managers it has constructed and returns an existing instance if one exists for the specified configuration file. As the threads utilize the same resources, there will quite probably be some resource conflicts. Database connection is a shared resource. If two threads want to access the database at the same time, one will either have to wait until the persistence layer has finished serving the first thread, or a pool of connections can be used.

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

Since database records are a shared resource, if a thread modifies a record, it is locked. If another thread tries to modify the record at the same time, then the result depends on the particular database implementation. Most probably an exception will be thrown by the ODBC driver. In case there is need for a Persistence Manager not shared with any other threads, the method *PersistenceManagerFactory.createNew* can be used. This method does not use the cached Persistence Managers, but creates a new one, that will not be given to other callers.

4.22 Logging

The persistence layer makes heavy use of logging. Every performed action is logged. Logging enables debugging of the application, to examination of unexpected behavior and auditing. It is possible to configure logging at runtime. Editing a logging configuration file can control logging behaviour.

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

The levels are:

DEBUG	Logging every operation detail. This level should be used only for debugging purposes, as the output is extremely abundant and can amount to thousands of entries in a few calls to a Persistence Manager instances.
INFO	Logging application progress - what operations are performed, what values stored, what data retrieved. This level should be used if there is need for auditing information. The output is nowhere near as abundant as on the previous level, but as all stored and retrieved values are logged, it can still reach undesirable quantities.
WARN	Logging potentially harmful situations that do not have affect on program flow.
ERROR	Logging error situations that interrupt the current program flow, but can be possible to recover from.
FATAL	Logging unrecoverable error situations that make it impossible for the application to continue. Setting the logging level to FATAL practically disables logging, as fatal errors are extremely rare under normal circumstances.

Chapter 5

The Persistence Manager Pattern

5.1 Context

Persistence Manager Pattern is necessary for developing an application that needs to store the objects it works with. Most probably, the persistence mechanism used is a relational database.

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

5.2 Problem

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

5.3 Motivation

- The most commonly used persistence mechanism at the moment is a relational database management system [BTR01].
- Every database vendor provides a slightly different syntax for functionality outside the standard SQL, which gives greater power to the programmer, but enforces the use of a proprietary technology.

- Object databases are relatively new and uncommon, and do not have the support relational databases have [BTR01].
- Embedding statements for data storage (e.g. SQL queries) in persistent classes is fast, both in performance and initial development, but hinders portability and forces reworking of the classes in case of changes in the data structure. For example, changing the field type of a class triggers a change of both the database and the persistence logic. In another instance, a database table column's data type is changed from INTEGER to BIT, but the object's field type remains a boolean.
- Adding persistence into the classes themselves is not always possible - they might be third party components.
- Persistence methodology is often very similar across applications, differing only in the exact data structure, which makes it a candidate for automation and reuse.

Developing a generic, reusable persistence mechanism takes a lot more effort and skills than embedding an application-specific mechanism in the application.

5.4 Solution

Create a component that is able to provide application objects with persistence. The component is not directly associated with the persistent classes and is therefore fully reusable. Configuring the component to handle new persistent classes is dynamical, done at run-time, from a configuration file. The component's functionality equals that of a relational database, providing a query mechanism, relational integrity, and transaction support, which are significant in many applications. The persistence provided is transparent

- the client programmers do not need to be aware of how class instances are made to persist. All the client programmer is aware of is that calling the methods of the component - e.g. store, delete, and retrieve - provides access to persistence. Extending the software to support other kinds of data stores besides relational databases should be possible with no impact on existing code.

5.5 Overview of the functionality of the Persistence Manager

In this section we list the functionality of Persistence Manager classes.

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

[*] Persistent classes can inherit from other persistent classes, extending them with new attributes. This inheritance needs to be supported in a relational database as well. This can be implemented either storing by the entire inheritance tree in one table, storing each class in a separate table including all the attributes, or storing each class in a separate table including only the attributes specific to that concrete class. When retrieving an object of a persistent class that has persistent extensions, the extensions will have to be queried as well [AMB00a].

5.6 Resulting Context

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

Our approach, however, has given rise to at least new problems.

New problem: performance overhead.

The persistence mechanism of the application has become notably more complicated, adding at least one, and possibly several, layers of logic between persistent classes and the data store. This incurs an additional performance load.

New problem: decrease in the functionality of the persistence mechanism.

With relational databases, the programmer can execute extremely complex queries just as readily as simple storage and retrieval statements. With a Persistence Manager, other workarounds must be used. Finding these work around can make an excellent thesis topic for new research.

5.7 Rationale

The general idea for a Persistence Manager-like architecture was introduced by Scott W. Ambler [AMB00b].

However, his work did not propose the broker in pattern form, and he specified the classes to be persistence aware.

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

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

Now that the Persistence Manager is ready, it can and will be used in many applications. The application programmer can utilize it without paying attention to or even being aware of whether the underlying data store is a database from the same vendor, from another

vendor, or an altogether different mechanism, such as an XML file. If a minor change is made in the underlying data store, the programmer need not change any part of the application; if a bigger change is made then there is no persistence logic code to change, only the Persistence Manager configuration file.

Chapter 6
Implementation Of DPL

There are three fundamental ideas underlying the implementation of DPL.

A) Every persistent object has a symbolic object id(OID), which is a string and is unique over classes. There is a mapping between persistent objects in memory and their OID.

B) References to persistent objects are encapsulated in classes called ONDEMAND. This enables reference cycles without "endless pain", and it enables intelligent loading and caching. The normal ONDEMAND does – as the name implies load the object in case of de-referencing with the help of the OID.

C) Every class is able to write and read itself to from a stream. That stream is storage independent and is used from the storage dependent layer to store the object.

In terms of class design, this means:

- Every Class that should be saved on a Persistence Medium has to be derived from class PERSISTENT.
- Every Class that should be saved on a Persistence Medium has to be referenced by special smart pointers called ONDEMAND.

There is one interface to the persistence mechanisms called PERSISTENCE-MANAGER.

There are two Phases involved in creating a new class with persistence:

A. The storage independent steps

B. The storage dependent steps (This needs to be implemented as part of future work).

First, the storage independent steps will be explained on an example:

The class COMPANY has several MACHINES. The corresponding ondemands are ONDEMAND_MACHINE and ONDEMAND_COMPANY.

6.1 Storage Independent Steps

6.1.1inherit from persistence and provide the features needed

The inheritence is applied as in the following examples :

```
--------- sourcecode start
class COMPANY                                   // Make Persistence
        inherit PERSISTENCE
                rename make as make_persistent      // Providing the features
                redefine write_to_stream, read_from_stream end
--------- sourcecode end
```

To model the has-relationship, we use a LINKED _LIST, one subclass of a more general CONTAINER CLASS, CONTAINER of ONDEMAND[MACHINES].

To make things more interesting, we use an expanded type. So we do not forget to create it. But on the other hand things are a little bit more complicated when we read in the object, as you will be discussed see in the explanation of 'read_from_stream' below.

```
--------- sourcecode start
machines: expanded LINKED_LIST[ONDEMAND _MACHINE]
--------- sourcecode end
```

The stream provides some features that can handle ondemands and containers of ondemand. But, there is still a need for an assignment attempt, since the stream reads only plain ONDEMANDs and the COMPANY uses ONDEMAND_MACHINEs. But since they are derived from ONDEMAND, the assignment attempt always works. The call stream.last_container returns a CONTAINER[ONDEMANDs].

In the COMPANY, the machines are a LINKED_LIST[ONDEMAND_MACHINE].

There is one possiblity to copy the CONTAINER to another COLLECTION. This the feature fill.

A temporary 'm' is needed, since 'machines' is an expanded type and the assignment is not allowed to expanded types.

```
--------- sourcecode start
  read_from_stream(stream: INPUT_OBJECT_STREAM) is
        local
          m: CONTAINER[ONDEMAND_MACHINE]
        do
          steam.read_ondemand_container
          ---cast ONDEMAND to ONDEMAND_MACHINE
```

```
        m ?= stream.list_ondemand_container
        machines.fill(m)
      end
--------- sourcecode end
```

Because we also want to save the Company, we have to provide a 'write_to_stream' feature, as in:

```
--------- sourcecode start
  write_to_stream(stream: OUTPUT_OBJECT_STREAM) is
      do
        precursor(stream)
        stream.append_ondemand_container(machines)
      end
--------- sourcecode end
```

First we have to call the precursor. This is important, because the class information and the OID are written to the stream in the precursor. Then we write all persistent members to the stream. For this purpose, the stream provides a number of features. It is vital to ensure, that the order of writing the persistent members is exactly the same as the order of reading the members! Otherwise, errors will occur during the reading the objects. Another feature named 'forward_ondemand' has to be defined. This feature will forward recursive calls during saving of object structures.

6.1.2 Provide the factory

To be able to read and especially create objects that come from the "outside" of the running process, we have to provide a class that knows how to do that creation. This is the factory. In addition, it knows how to create ONDEMANDS that belong to the class the factory is responsible for.

It is always the same and coded as:

```
--------- sourcecode start
class COMPANY_FACTORY
inherit
  FACTORY
feature
  create_object(stream: INPUT_OBJECT_STREAM): COMPANY is
        do
          !COMPANY! result.create_from_stream(stream)  // Creating Object
        end
end – class COMPANY_FACTORY
--------- sourcecode end
```

6.1.3 Provide the ondemand

The ondemand classes, which are used to reference persistent objects, are derived from ONDEMAND. They should be named "ONDEMAND_ClassName". They look like:

```
--------- sourcecode start
class ONDEMAND_MACHINE
inherit
  ONDEMAND
        redefined reference end
creation make_from_oid, make_from_reference
feature
  referenece: MACHINE
end – class ONDEMAND_MACHINE
--------- sourcecode end
```

6.1.4 Registering the persistent classes with the persistence_manager

To register the persistent classes within your application, first thing is to inherit the interface to the persistence manager.

```
--------- sourcecode start
class APPLICATION
inherit
  PERSISTENCE_MANAGER
feature
  register_persistent_classes is
        local
          f: FACTORY
          f_os: OBJECT_SOURCE
```

```
        do
          !FILE_OBJECT_SOURCE! f_os
          --MACHINE
          !MACHINE_FACTORY! f
          db.register_type("MACINE", "ONDEMAND_MACHINE", f, f_os)
          --COMPANY
          !COMPANY_FACTORY! f
          db.register_type("COMPANY", "ONDEMAND_COMPANY", f, f_os)
        end
--------- sourcecode end
```

This is achieved by deriving from "PERSISTENCE_MANAGER". The feature 'db' returns the "OBJECT_SOURCE_MANAGER". The registration tells the object_source_manager, which class has the appropriate ondemand_class, factory and object source. The OBJECT_SOURCE will be explained later. It is dependent on the storage (file, odbc). It can be changed without changing the rest of the DPL.

In the following example, the registration is bundled in its own feature. It is important to the registration of all persistent classes before the persistence layer is used.

6.2 Usage Of DPL

After having written all the classes and features for a persistent class, one may ask how to use the persistence layer within an application. There are only few things you can do; basically, storing and retrieving objects from the store. We discuss these below.

6.3 When an object needs to be saved?

An object that was read from store and not changed does not need to be stored! Only after having changed the object is a save needed. An object that needs saving is called “dirty”. To mark an object for saving when the next call to ‘store’ is done, the feature “set_dirty” should be called. Usually, every change to an object should also do a call to set_dirty. If an object is created inside the process by the creation procedure of PERSISTENT, it is marked dirty automatically.

6.3.1 Storing objects

To store an object ‘o’ of persistent class A, just call the feature ‘store’ on ‘o’. This will result in saving all dirty objects that ‘o’ depends on.

In the example, a call to c.store (c:COMPANY) would save ‘c’ itself and the machines the company has (only the dirty ones).

6.3.2 Retrieving objects

There are methods for retrieving objects. First we can load an object by an OID. The second method is to load all objects of a specific class. After having loaded an object, in terms of "class" you hold a PERSISTENT in your hands. An assignment attempt will be used to make it useful for you.

6.4 Summary

In this chapter, we have discuss how to make an object persistent using Durable Persistence Layer (DPL). The complete source code of DPL has not been included in this thesis. In order to view the source code, contact Dr. R. D. Kent*.

*rkent@uwindsor.ca

Chapter 7

Testing and Post-analysis

7.1 DPL Compared to Embedding SQL Directly

This section deals with Testing and Post-analysis.

A simple approach to persistence logic is embedding SQL statements directly in application code. Most of the advantages of a Persistence Manager were discussed in sections 5.5 and 5.6, Persistence Manager pattern.

7.1.1 Sample Code

Simple Retrieval

The following two sections of code both retrieve all Simple Persons from the database that have a location of 'Windsor'. When comparing the sections, we can see that using DPL yields a much more terse and understandable program code, not to mention with less effort then using embedded SQL.

Example 7.1.2 Using embedded SQL for simple retrieval.

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

Example 7.1.3 Using DPL for simple retrieval.

```
Simplepatient param = new SimplePatient();
param.location = "Windsor";
Query query = createQuery(param, new String[ ]{"location"}, null);
Collection results = retrieveCollection(query);
```

7.2 Sample Code

Simple storage

The following two examples of code both store a new SimplePerson in the database with DPL this action is even more simple than retrieval.

Example 7.2.1 Simple Storage

```
Connection connection = DriverManager.getConnection("jdbc:odbc:emt");
Statement stmt = connection.createStatement();
SimplePatient person = new SimplePatient();
ResultSet rs = stmt.executeQuery(
"SELECT MAX(id) + 1 AS newId FROM SIMPLEPATIENT");
rs.next();
// The new id value is got this value to be able to set it to the
// object immediately.
person.id = rs.getInt("newId");
rs.close();
person.firstName = "John";
person.lastName = "Doe";
person.location = "Windsor";
person.phone = "253-3000";
stmt.executeUpdate(
"INSERT INTO SIMPLEPATIENT (id,firstName,lastName,location,phone)" +
        "VALUES (" + person.id + ", '" + person.firstName + "', '" +
         person.lastName + "', '" + person.location + "', '" +
        person.phone + "')");
stmt.close();
connection.close();
```

Example 7.2.2 Using DPL

```
SimplePatient person = new SimplePatient();
person.firstName = "John";
person.lastName = "Doe";
person.location = "Windsor";
person.phone = "253-3000";
store(person);
```

7.3 Tradeoffs

The initial disadvantage of implementing the Persistence Manager pattern is that it is a non-trivial task, possibly greater than developing the application itself. However, as the component is easily reusable in consecutive applications, it pays off in the manner in which it supports re-use.

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

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

7.4 Benchmark Information

Test environment

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

Computer: P4, 766 MHZ, 256MB RAM, Windows 98

Database system: Microsoft Access 97

Database driver: Access Driver

Results Table

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

Repeated	Ops	X		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	

The following operations were benchmarked:

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

7.5 Benchmarks

Action 1 *- Find all Simplepatient from the location 'Windsor'.*

Repeated?	Ops	Embedded SQL		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	450	4500	10.00	5400	12.00	1.20
Yes		1850	4.11	1500	3.33	0.81

Action 2 *- update all Simplepatient retrieved in action 1.*

Repeated?	Ops	Embedded SQL		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	450	3350	7.44	3450	7.67	1.03
Yes		1650	3.67	3300	7.33	2.00

Action 3- Find all Persons from the location 'London' and retrieve their Phones automatically

Repeated?	Ops	Embedded SQL		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	50	8450	169.00	8750	175.00	1.04
Yes		5650	113.00	960	19.20	0.17

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

Repeated?	Ops	Embedded SQL		DPL		Factor to DPL
		Sum [ms]	Avg [ms]	Sum [ms]	Avg [ms]	
No	50	2200	44.00	4850	97.00	2.20

When examining the results, we can see that the main strength of DPL lies in retrieval in some cases it can be very fast, e.g. when executing repeated retrievals in action 3.

Caching causes this good performance - as objects that are cached do not need to be fully read from the database (once their identity is clear, their cached instance can be reused), several operations could be skipped altogether. An especially high performance yield comes with relationships - cached objects already have their related objects set in Place, so additional queries to the database for getting the related objects are not needed. On the whole, the results were surprisingly good. We did not expect all the application logic existent in DPL to have such a small time overhead, and feared at least a triple time penalty with using DPL.

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

7.6 DPL Compared to Similar Components

There is abundant software available that performs mapping objects to relational databases. Some products are quite powerful, providing a rich user interface for configuration and offering automatic generation of the class diagram from the database table diagram. However, most products do not have the level of unintrusiveness that the Persistence Manager pattern, and consequently the Durable persistence layer, have.

Persistent classes usually have to extend a superclass that couples them with the persistence layer. There is only one component that does compare to DPL in terms of unintrusiveness and functionality - ObjectRelationalBridge.

7.6.1 Object Relational Bridge

ObjectRelationalBridge (OJB) is an open source software (publicly available at *(http://objectbridge.sourceforge.net/)* roughly following the Persistence Manager pattern. Its functionality and structure is quite similar to DPL. In the following sub section we compare the advantages and disadvantages of these approaches.

7.6.2 Advantages of OJB Compared to DPL:

The following are a partial listing of OJB advantages:

- supports cursors (for information on cursors, see 8.2 Further Development)
- has partial support for transactions - only if the underlying database supports transactions. Some databases do not support transactions, like MySQL (http://www.mysql.com/)
- supports virtual proxies (for information on virtual proxies, see 8.2 Further Development)
- supports extent classes (for information on extent classes, see 8.2 Further Development)

7.6.3 Disadvantages of OJB Compared to DPL:

- Error handling is severely lacking - in most cases, encountered exceptions are caught and not permitted to rise outside the component, thus giving no clue if there was an error.
- Has virtually no logging. It is possible to have some messages printed to the console after recompiling the classes to support this, but the messages are non-expressive and few.
- Does not support composite identities.

- Table columns can only have certain pre-defined types - the types are hard-coded in. Although they cover most of the possible column types, they do not cover all of them.
- Object identity fields are hard-coded to integer type.
- There is only one identity generation strategy - using a sequence table in the database.
- It is possible to configure OJB to use an arbitrary strategy, but not on a per-class basis or even a per-Persistence Manager basis; the specified strategy will then be used by all Persistence Managers. Configuring this arbitrary strategy implementation has to be done outside the application - the broker configuration file holds no information for it.
- There is one global object cache for the entire machine. As caches do not know where an object is from - they only know its type and identity values, this can lead to problems in cases where there are multiple applications running on the same machine that use the same persistent classes, but store them in different data stores.
- Contains some bugs - for example, one type of query disregards its criteria and returns every object for the specified class it can find. *(http://jakarta.apache.org/ojb/)*
- Instantiating persistent classes is not flexible - the user cannot specify an arbitrary object creation strategy.
- Does not support custom SQL.
- Does not support a custom way for accessing object fields - the access is hard-coded to be done using reflection and cannot be changed.

- API documentation is inadequate, incomplete and defective.

7.7 Benchmarks

Action 1- Find all Simplepatient from the location 'Windsor'.

Repeated	Ops	OJB		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	450	6550	14.56	5400	12.00	0.82
Yes		3200	7.11	1500	3.33	0.47

Action 2- Update all Simplepatient retrieved in action 1.

Repeated	Ops	OJB		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	450	10750	23.89	3450	7.67	0.32
Yes		10350	23.00	3300	7.33	0.32

Action 3- Find all Persons from the location 'London' and retrieve their Phones automatically.

Repeated	Ops	OJB		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	50	14550	291.00	8750	175.00	0.60
Yes		1650	33.00	960	19.20	0.58

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

Repeated	Ops	OJB		DPL		Factor to DPL
		Sum[ms]	Avg[ms]	Sum[ms]	Avg[ms]	
No	50	10550	210.00	4850	97.00	0.46

These benchmark results are even more surprising than those in comparison with embedded SQL. I expected OJB, a product that has had a considerably longer lifetime, to be much better in terms of performance than DPL, a product that is a newborn. It is possible that such good relative performance of DPL is caused by some issues that need to be handled but have not occurred to me.

On the basis of these results, DPL can be pronounced to give excellent performance. The Persistence Manager pattern is a good approach to handling data persistence. As we saw from its comparison with directly embedded SQL, it does not incur an unacceptable performance .In many cases, a Persistence Manager can give results fast, because it caches the persistent objects it encountered.

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

On the basis of these results, we can conclude that the Persistence Manager is a good approach to handling data persistence, although not a solution fit for every application. When comparing DPL, one implementation of the Persistence Manager pattern, to another implementation, OJB, we see that the main strength of DPL is in its careful design and flexibility - it has powerful logging, good error handling and its architecture is

compartmentalized so that custom sub-components can be used with the basic aspects of persistence operations, like creating SQL syntax and accessing object fields.

At the present time, DPL is lacking some useful functionality that OJB possesses, such as supporting cursors and transactions. These items have been noted on the list for future developments. These issues do not impact negatively on the finding of this thesis, however.

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

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

Chapter 8
Conclusion

8.1 Conclusion

In this thesis, we set out to achieve the following goals:

To propose a reusable generic approach to handling persistence logic - the Persistence Manager design pattern, that solves several problems when using relational database management systems in an object-oriented application environment. The basic idea for such software was introduced in Scott W.Ambler's white paper "Mapping Objects to Relational Databases" [Amb00a].

To present DPL, a component we developed in the Eiffel programming language as an implementation of this pattern. To compare DPL performance to the most basic approach to persistence logic to find out whether the Persistence Manager offers a more preferable solution; the most basic approach being embedding database access logic directly into application code. We produced detail architecture of DPL showing in figure 4.13.

DPL was also compared to other software products that follow the Persistence Manager pattern, to find out whether DPL is a viable solution among similar components.

All the three goals were realized. Based on the results of the comparison with the basic approach, we deemed the Persistence Manager to be a well-designed approach to

persistence logic. Using DPL yielded faster program development, better modularized application structure, and in cases even better performance on persistence operations.

8.2 Future Development

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

Overview of items scheduled for further development:

Development Item	Details	Priority
Cursors	Provide support for a cursor that points to a set of retrieved objects and that the user can move to the next or previous retrieved object. The cursor makes use of lazy initialization - an object is retrieved only when the cursor arrives at the object.	Primary
Transaction	Provide support for transactions – grouping persistence operations into atomic units that succeeds or fails as a whole. Transactions are very helpful in many applications, especially in the business realm (e.g. making a bank transfer consists of several operations - decreasing the amount on the transferor side and increasing on the transferee side - that must all either succeed or fail).	Primary
Object Query Language	Provide support for the Object Query Language (OQL). OQL is an SQL-like declarative Languages that provides a rich environment for efficient querying of database objects, including high-level primitives for object sets and structures [ODMG1]. OQL has the advantage of simplicity and better overview, when compared to constructing queries by setting parameters and criteria via method calls.	Secondary
Virtual proxies	Provide support for lazy initialization of persistent objects via virtual proxies. A virtual proxy is a proxy for another object (the real persistent subject) that materializes the subject when it is first referenced [Lar01]. As it could happen that the real object might never really be needed, this can provide a notable performance gain.	Secondary

Development Item	Details	Priority
Distributed use	Provide support for distributed use, in which there is a central Persistence Manager server that multiple clients connect to and execute persistence operations on.	tertiary
Logging configurable In the same configuration as persistence	Allow logging configuration to be specified in the persistence configuration file.	optional
XML files as data stores	Provide support for using XML files as data storage mechanisms. Although XML files are not comparable with databases in terms of ease of use, performance and functionality, they have the benefit of simplicity - there is no need for a database management system, everything is contained in files. For smaller projects, they provide adequate Performance.	under consid-eration
Graphical user interface for configuration	Provide a graphical application for producing It should be able to generate lowest configuration database table structure and a configuration file from the class structure. Another approach would be to use an existing graphical tool (such do exist - Rational Rose is one) and add support for DPL via plugins or scripting.	lowest

Current programming languages are very good at manipulating objects, which only exist during the execution time of the program. There are a multitude of applications, however, that needs data to last "forever". Applications are becoming more complex by the day. Programmers should not have to concentrate on changing the form of the data, but on the application. DPL provides a vehicle to do this.

The software market has very few software products available that follow the Persistence Manager concept strictly. Among the similar available software, DPL performed very well. Its functionality should be extended, and the missing functionalities have already been planned into future development.

References

[AMB02] Ambler, S.W. (2002a). "*A Class Type Architecture For Layering Your Application,*" A Ronin International White Paper. posted at www.ronin-intl.com

[AMB01] Ambler, S.W. (2001a). The Object Primer 2nd Edition: "*The Application Developer's Guide to Object Orientation.*" New York: Cambridge University Press. www.ambysoft.com/theObjectPrimer.html

[AMB00a] Scott W. Ambler (October 2000). "*Mapping Objects To Relational Databases, An AmbySoft Inc.*" *White papers* ,http://www.AmbySoft.com/mappingObjects.pdf

[AMB00b] Scott W. Ambler (November 2000). "*The Design of a Robust Persistence Layer For Relational Databases An AmbySoft Inc.*" *White Papers.* http://www.ambysoft.com/persistenceLayer.pdf

[AMB98] 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

[ABL96] B. Liskov, A. Adya, M. Castro, M. Day, R. Gruber, U. Maheshwari, A. Myers, L. Shrira. *"Safe and Efficient Sharing of Persistent Objects in Thor"* Proceedings of SIGMOD, Montreal, Canada, June 1996. ACM SIGMOD Record 25(2): 318–329.

[ACS00] Ambler, S.W. & Constantine, L.L. (2000). "*The Unified Process Inception Phas.*" Gilroy, CA: CMP Books. http://www.ambysoft.com/inceptionPhase.html

[ADK01] Kirby, GNC, Dearle, A, Sjøberg, D (eds), "*Persistent Object Systems: Design, Implementation and Use.*" Vol. 2135, Springer, ISBN 3-540-42735-X. 2001. Proc. 9th International Workshop on Persistent Object Systems, Lillehammer, Norway, 2001, (POS9).

[ADH00] Dearle, A, Hulse, D. In: *"Operating System Support for Persistent Systems: Past, Present and Future."* Software - Practice and Experience, Special Issue on Persistent Object Systems 30, 4, pp 295-324. 2000.

[ADH99] O'Lenskie, A, Dearle, A, Hulse, D. In: "*Persistent Operating System Support for Persistent CORBA Objects.*" Advances in Persistent Object Systems 1999.

[AMR99] R, Jordan, M, Atkinson, MP (eds), *Proc. 8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3), Tiburon, California, 1998, pp 92-111.* Morgan Kaufmann, ISBN 1-55860-585-1. 1999.

[AGE 99] Ole Agesen, "*Space and Time-Efficient Hashing of Garbage-Collected Object*," journal, Theory and Practice of Object Systems, volume 5, number 2, pages 119-124, year 1999, http://www.sunlabs.com/research/java-topics/pubs/99-tapos.ps

[BAG91], Bagherzadeh, Nader and Heng, S-I. and Wu,"*A Parallel Asynchronous Garbage Collection Algorithm for Distributed Systems*" Journal "IEEE Transactions on Knowledge and Data Engineering," publisher IEEE, volume3, number1, month mar, pages "100-107", year 1991.

[BDM00a] Morrison, R, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Munro, DS, Warboys, B. In: "*An Approach to Compliance in Software Architectures:*" IEE Computing & Control Engineering Journal, Special Issue on Informatics 11, 4, pp 195-200, 2000.

[BMN 02] Michael L. Nelson, B. Danette Allen, "*Object Persistence and Availability in Digital Libraries*," NASA Langley Research Center Hampton, VA 23681, D-libMagazine, January 2002, Volume8, Number 1, ISSN 1082-9873 http://www.dlib.org/dlib/january02.

[BDM00b] Morrison, R, Balasubramaniam, D, Greenwood, RM, Kirby, GNC, Mayes, K, Munro, DS, Warboys, BC. In: "*A Compliant Persistent Architecture.*" Software - Practice and Experience, Special Issue on Persistent Object Systems 30, 4, pp 363-386, 2000.

[BMR01] Bertrand Meyer."*Invitation to Eiffel*," ISE Technical Report TR-EI-67/IV. July 2001, ISE Eiffel environment. http://www.eiffel.com

[BFM96] 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.

[BMR92] Bertrand Meyer. *Eiffel: "The Language (Second Printing)*," Prentice Hall International, 1992. ISBN 0-13-247925-7, P20-44.

[BTR01] Biblio Tech Review (April 2001). Database Management Systems (DBMS). Biblio Tech Review, Technical Briefings,http://www.biblio-tech.com/html/databases.html

[CGL01] Craig Larman (July 2001). *Applying UML and Patterns: "An Introduction to Object-Oriented Analysis and Design and the Unified Process*." Addison-Wesley.

[GLR91] L. Gunaseelan, R.J. LeBlanc: Distributed Eiffel: *A Language for Programming Multi-granular Distributed Objects on the Clouds Operating System*. Report 91/50, Georgia Institute of Technology, College of Computing, 1991, Revised version in proceedings of the Fourth International Conference on Computer Languages (ICCL), IEEE Computer Society, San Francisco, Calif., April 1992.

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

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

Glossary

Business/domain class. Used to implement the concepts pertinent to your business domain, such as "customer" or "product." Business/domain classes are usually found during the analysis process. Although business/domain classes often focus on the data aspects of your business objects, they will also implement methods specific to the individual business concept [AMB02]

Class-type architecture. A defined approach to layering the classes that comprise the software of a system. The interaction between classes is often restricted based on the layer to which they belong [AMB02]

Client/server (C/S) architecture A computing environment that satisfies the business need by appropriately allocating the application processing between the client and the server processes [AMB01]

Concurrency The issues involved with allowing multiple people simultaneous access to your persistence mechanism.

Coupling A measure of how connected two items are.

CRUD Acronym for create, retrieve, update, delete. The basic functionality that a persistence mechanism must support.

Database server A server which has a database installed on it.

Distributed objects An object-oriented architecture in which objects running in separate memory spaces (i.e. different computers) interact with one another transparently.

Layering The organization of software collections (layers) of classes or components that fulfill a common purpose.

Lock An indication that a table, record, class, object, ... is reserved so that work can be accomplished on the item being locked. A lock is established, the work is done, and the lock is removed.

Object database (ODB) A permanent storage mechanism, also known as an objectbase or an object-oriented database management system (OODBMS), which natively supports the persistence of objects.[AMB00b]

OOCRUD Object-oriented create, retrieve, update, and delete

Object identifiers (OIDs) A unique identifier assigned to objects, typically a large integer number. OIDs are the object-oriented equivalent of keys in the relational world [AMB00b].

Permanent storage Any physical medium to which data can be saved, retrieved, and deleted. Potential permanent storage mechanisms for objects include relational databases, files, and object databases.

Persistence class Provide the capability to store objects permanently. By encapsulating the storage and retrieval of objects via persistence classes, you are able to use various storage technologies interchangeably without affecting your applications.

Portability A measure of how easy it is to move an application to another environment (which may vary by the configuration of either their software and hardware). The easier it is to move an application to another environment, the more portable we say that application is.

Persistence mechanism The permanent storage facility used to make objects persistent. Examples include relational databases, object databases, flat files, and object/relational databases.

Pessimistic locking An approach to concurrency in which an item is locked 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 whereas the item is in memory, but at the same time is disallows others to work with it while someone else does [AMB00b].

Persistence layer A collection of classes that provides objects the ability to be persistent, being effectively a wrapper for the persistence mechanism [Amb00b]

Persistence logic Application code that handles storing, deleting and retrieving persistent data

Persistent data Data that has persistence

Persistence operations Operations like saving, deleting and retrieving

Relational database (RDB) A permanent storage mechanism in which data is stored as rows in tables. RDBs don't natively support the persistence of objects, requiring the additional work on the part of developers and/or the use of a persistence layer.

Sequence diagram A UML diagram that models the sequential logic, in effect, the time ordering of messages between objects [CGL01]

SQL Structured Query Language, a standard mechanism used to CRUD records in a relational database.

Soft reference An object reference that is cleared at the discretion of the garbage collector in response to memory demand

Transaction A transaction is a single unit of work performed in a persistence mechanism. A transaction may be one or more updates to a persistence mechanism, one or more reads, one or more deletes, or any combination thereof.

Transparent persistence Automatically provided persistence without special effort on the client programmer side. No difference between persistent and transient objects.

User interface class A class that provides the capability for users to interact with the system. User interface classes typically define a graphical user interface for an application, although other interface styles, such as voice command or HTML, are also implemented via user-interface classes [AMB02]

Appendix A
Sample Persistence Classes

```
---------------
--Main Class--
---------------
class PE
inherit
        PERSISTENCE_MANAGER
        MEMORY
        FILE_OBJECT_SOURCE
creation
make
feature
        objectID: STRING
        human: HUMAN
        man: MAN
        woman: WOMAN
        car: CAR
        fos: FILE_OBJECT_SOURCE
        ob_type: STRING
        ---------------
        make is
        ---------------
                do
                        io.new_line
                        io.put_string("%N*************************************")
                        io.put_string("%NWelcome to the world of PERSISTENCY")
                        io.put_string("%N*************************************")
------------------------------
-- Registering Persistent classes
------------------------------
                        io.put_string("%N%NRegistering persistent classes....!")
                        register_persistent_classes
                --      !!human.make
------------------------------
-- Restoring the Object Structure
------------------------------


-----------------------
-- Restoration of HUMAN
-----------------------
                        io.new_line
                        io.put_string("--------------- Trying to restore HUMAN")
                        ob_type := "HUMAN"
                        objectID :=     Read_Object_From_File(ob_type)
                        human ?= db.get_object_by_oid(objectID, ob_type)


                        if human = void then
                                io.new_line
                                io.put_string("--------------- Creating NEW HUMAN!")
```

```
                    !!human.make
                    io.put_string("%NHUMAN is set up")
            else
                    io.put_string("%N--------------- Restoration of HUMAN Done!%N")
                    io.put_string("%NThe value of HUMAN after restoration is:%N")
                    io.put_string(human.data.out)
            end

--------------------
-- Remaining Execution
--------------------
            io.put_string("%NExecution completed!")
        end
-------------------------
feature -- persistence
-------------------------
  register_persistent_classes is

        local
       f: FACTORY
            os: OBJECT_SOURCE
    do
            !FILE_OBJECT_SOURCE!os
            !HUMAN_FACTORY!f
            db.register_type("HUMAN", "ONDEMAND_HUMAN", f, os)
            !MAN_FACTORY!f
            db.register_type("MAN", "ONDEMAND_MAN", f, os)
            !WOMAN_FACTORY!f
            db.register_type("WOMAN", "ONDEMAND_WOMAN", f, os)
            !CAR_FACTORY!f
            db.register_type("CAR", "ONDEMAND_CAR", f, os)
            io.put_string("%NRegistration done!")
    end
end -- class PE
```

```
---------------------
--Class HUMAN --
---------------------
Class HUMAN
inherit
        PERSISTENT
        rename make as make_persistent
        redefine write_to_stream, read_from_stream, create_from_stream
        end

creation
        make, create_from_stream

feature
        gen: CHARACTER
        gender: STRING
        name: STRING
        data: STRING
        car: CAR
        man: MAN
        woman: WOMAN
        humans : expanded LINKED_LIST[ONDEMAND_HUMAN]
-------------------------------
        make is
-------------------------------
                do
                        make_persistent
                        data_for_man
                --      execute
                        set_dirty
                        io.put_string("%NStoring the Object References for HUMAN... ")
                        store
                        io.put_string("%NHUMAN Stored!%N")
                        io.put_string("Executing the remaining program....%N")
                end

-------------------------------
        execute is
-------------------------------
                do
                        io.new_line
                        io.putstring("[M/m = MALE; F/f = FEMALE.....]%N")
                        io.putstring("Enter Your gender HUMAN: ")
                        io.readchar
                        gen := io.lastchar
                        check_gender
                end
-------------------------------
        check_gender is
-------------------------------
                do
                        inspect gen
                                when 'M', 'm' then data_for_man
                                when 'F', 'f' then data_for_woman
                                else
                                        io.putstring("Wrong entry...!")
```

```
                                        io.putstring("%NPlease enter the data again: ")
                                        io.readchar
                                        gen := io.lastchar
                                        check_gender
                              end
                    end
----------------------------------
          data_for_man is
----------------------------------
          do
                    name := "Kamran"
                    data := "HUMAN 1975-03-29- choudhe@uwindsor.ca"
                    io.put_string("%NName of Human: %N" + name)
                    io.new_line
                    io.put_string("Data of Human: %N" + data)
          end
----------------------------------
          data_for_woman is
----------------------------------
          do
                    name := "Jennifer Lopez"
                    data := "HUMAN Jennifer@uwindsor.ca"
                    io.put_string("Name of Human: %N" + name)
                    io.new_line
                    io.put_string("Data of Human: %N" + data)
          end
--------------------------------------------
-- READ FROM STREAM --
--------------------------------------------
          read_from_stream(stream: INPUT_OBJECT_STREAM) is

          --        local

                    --        h: CONTAINER[ONDEMAND_HUMAN]

                    do
                    --        stream.read_ondemand_container
                    --        h ?= stream.last_ondemand_container

                              stream.read_string
                              name := stream.last_string

                              stream.read_string
                              data := stream.last_string

                    --        !!humans.make
                    --        humans.fill(h)


          end
```

```
----------------------------------------
WRITE TO STREAM
----------------------------------------
        write_to_stream(stream: OUTPUT_OBJECT_STREAM) is
                do
                        precursor(stream)
                --      stream.append_ondemand_container(humans)
                        stream.append_string(name)
                        stream.append_string(data)
                end

----------------------------------------
-- FORWARD ON DEMAND --
----------------------------------------

        forward_ondemands(f: FUNCTOR; condition: BOOLEAN) is

                do
                        forward_ondemand_container(humans, f, condition)
                end

----------------------------------------
-- CREATE FROM STREAM --
----------------------------------------

create_from_stream(stream: INPUT_OBJECT_STREAM) is

        do
                io.put_string("%NThis is the return of HUMAN!")
                precursor(stream)
        end


        end -- class HUMAN
```

```
------------------
-- Class MAN --
------------------

class MAN
inherit
        PERSISTENT
        rename make as make_persistent
        redefine write_to_stream, read_from_stream
        end
creation
        make, get_data, create_from_stream
feature
        name, country: STRING
        mans : expanded LINKED_LIST[ONDEMAND_MAN]
----------------
        make is
----------------
                do
                        make_persistent
                        get_data
                        set_dirty
                        io.put_string("%NStoring the Object References for MAN...")
                        store
                        io.put_string("%NMAN Stored!%N")
                        io.put_string("Executing the remaining program....%N")
                end
----------------------------------------
        get_data is
----------------------------------------
                do
                        name := "john"
                        io.putstring("Enter your Country " + name + ": ")
                        io.readword
                        country := io.laststring
                        io.putstring("Welcome to my world " + name + ".")
                        io.new_line
                end
----------------------------------------
-- READ FROM STREAM --
----------------------------------------
        read_from_stream(stream: INPUT_OBJECT_STREAM) is
                local
                        m: CONTAINER[ONDEMAND_MAN]
                do
                        stream.read_string
                        name := stream.last_string
                        stream.read_string
                        country := stream.last_string
                        stream.read_ondemand_container
                        m ?= stream.last_ondemand_container
                        !!mans.make
                        mans.fill(m)
                end
```

```
---------------------------------------------
-- WRITE TO STREAM --
---------------------------------------------
        write_to_stream(stream: OUTPUT_OBJECT_STREAM) is
                do
                        precursor(stream)
                        stream.append_string(name)
                        stream.append_string(country)
                        stream.append_ondemand_container(mans)
                end
---------------------------------------------
-- FORWARD ON DEMAND --
---------------------------------------------
        forward_ondemands(f: FUNCTOR; condition: BOOLEAN) is
                do
                forward_ondemand_container(mans, f, condition)
                end
end -- class MAN
```

```
-----------------------
--Class WOMAN --
-----------------------

class WOMAN
inherit
    PERSISTENT
    rename make as make_persistent
    redefine write_to_stream, read_from_stream
    end
creation
    make, get_data, create_from_stream
feature
    name, country: STRING
    womans : expanded LINKED_LIST[ONDEMAND_WOMAN]
----------------
    make is
----------------
            do
                    make_persistent
                    get_data
                    set_dirty
                    io.put_string("%NStoring the Object References for WOMAN...")
                    store
                    io.put_string("%NWOMAN Stored!%N")
                    io.put_string("Executing the remaining program....%N")

            end
---------------------------------------
    get_data is
---------------------------------------
            do
                    io.new_line
                    name := "Jennifer"
                    io.putstring("Enter your country " + name + ": ")
                    io.readword
                    country := io.laststring
                    io.putstring("Welcome to my world " + name + ".")
                    io.new_line
            end
-----------------------------------------
-- READ FROM STREAM --
-----------------------------------------
    read_from_stream(stream: INPUT_OBJECT_STREAM) is
            local
                    w: CONTAINER[ONDEMAND_WOMAN]
            do
                    stream.read_string
                    name := stream.last_string
                    stream.read_string
                    country := stream.last_string
                    stream.read_ondemand_container
                    w ?= stream.last_ondemand_container
                    !!womans.make
                    womans.fill(w)
            end
```

```
----------------------------------------
-- WRITE TO STREAM --
----------------------------------------
    write_to_stream(stream: OUTPUT_OBJECT_STREAM) is
        do
                precursor(stream)
                stream.append_string(name)
                stream.append_string(country)
                stream.append_ondemand_container(womans)
        end
----------------------------------------
-- FORWARD ON DEMAND --
----------------------------------------

    forward_ondemands(f: FUNCTOR; condition: BOOLEAN) is
        do
                forward_ondemand_container(womans, f, condition)
        end
end -- class WOMAN
```

```
-----------------------
-- Class Car --
-----------------------

    class CAR
    inherit
        PERSISTENT
        rename make as make_persistent
        redefine write_to_stream, read_from_stream
        end
    creation
        get_data, make, create_from_stream
    feature
        model: STRING
        year: INTEGER
        cars : expanded LINKED_LIST[ONDEMAND_CAR]
    ----------------------
        make is
    ----------------------
                do
                        make_persistent
                        get_data
                        set_dirty
                        io.put_string("%NStoring the Object References for CAR... ")
                        store
                        io.put_string("%NCAR Stored!%N")
                        io.put_string("Executing the remaining program....%N")
                end
    ---------------------------
        get_data is
    ---------------------------
                do
                        model := "Mercedes- Benz"
                        year   := 2002
                end
    ---------------------------------------
    -- READ FROM STREAM --
    ---------------------------------------
        read_from_stream(stream: INPUT_OBJECT_STREAM) is

                local
                        c: CONTAINER[ONDEMAND_CAR]
                do
                        stream.read_string
                        model := stream.last_string
                        stream.read_integer
                        year := stream.last_integer
                        stream.read_ondemand_container
                        c ?= stream.last_ondemand_container
                        !!cars.make
                        cars.fill(c)
                end
```

```
------------------------------------------
-- WRITE TO STREAM --
------------------------------------------
    write_to_stream(stream: OUTPUT_OBJECT_STREAM) is
            do
                    precursor(stream)
                    stream.append_string(model)
                    stream.append_string(year.out)
                    stream.append_ondemand_container(cars)
            end
------------------------------------------
-- FORWARD ON DEMAND --
------------------------------------------
    forward_ondemands(f: FUNCTOR; condition: BOOLEAN) is
            do
                    forward_ondemand_container(cars, f, condition)
            end
end -- class CAR
```

Vita Auctoris

Kamran Choudhery was born in Karachi, a city in Pakistan on 27th August 1975. He graduated from Sir Syed University Of Engineering and Technology on March 1998 with a Bachelor of Science degree in Computer Engineering. In September 1999, Kamran joined the Master of Science program in Computer Science at the University of Windsor[1], Windsor, Ontario, Canada. Kamran is planning to pursue a Doctor of Philosophy Degree in Computer Science at the Computer Science Department at the University of Western Ontario[2], Ontario, Canada. Where he has been admitted and will be joining from September 2002.

[1] www.cs.uwindsor.ca

[2] www.cs.uwo.ca