E-YEARBOOK

(A JAVA-ORACLE DATABASE IMPLEMENTATION)

BY

FATI TINI BABA
PGD/MCS/2005/2006/1190

DEPARTMENT OF MATHEMATICS/COMPUTER SCIENCE
FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA

MARCH, 2007

E-YEARBOOK

(A JAVA-ORACLE DATABASE IMPLEMENTATION)

BY

FATI TINI BABA
PGD/MCS/2005/2006/1190

DEPARTMENT OF MATHEMATICS/COMPUTER SCIENCE
FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA

A PROJECT SUBMITTED TO THE SCHOOL OF SCIENCE AND SCIENCE EDUCATION

IN PARTIAL FULFILLMENT OF
THE REQUIREMENT FOR THE AWARD OF
POST-GRADUATE DIPLOMA COMPUTER SCIENCE,

MARCH, 2007

CERTIFICATION

This is to certify that this project work was carried out by **Baba Fati Tini** under the supervision of Mr. Victor Akinola of the Department of Mathematics/Computer Science in partial fulfillment of the requirement for the award of Post-Graduate Diploma in Computer Science.

18TH SEPTEMBER, 2007

Supervisor's Signature **Date**

Head of Department's Signature **Date**

External Examiner's Signature **Date**

DEDICATION

To Almighty Allah, beneficent, merciful and knowledgeable.

This work is dedicated to the memory of my late mother Hajia Hauwa'u Talatu Baba Nahannu Dama. May her soul rest in perfect peace.

ACKNOWLEDGEMENT

My profound Gratitude goes to the Almighty Allah, who has given me the strength and spared my life to complete this programme successfully.

I wish to express my deep appreciation to my supervisor, Mr. Victor Akinola, who has contributed to this project both explicitly and implicitly. His kindness and understanding of humanity is highly appreciated.

I equally wish to sincerely thank the Head of Mathematics/Computer Science Department and to all my Lecturers, especially, Mr Mohammed Jiya, Alhaji D. Hakimi, Prince Badmus, Mr C. Abraham, Mallam Salihu, Mr B. Gbolahan, Mallam Idris, Alh. Isah Audu, and Mallam Ndanusa, the coordinator, for imparting the needed knowledge towards the realization of this work.

Mallam Mohammed Usman Dattijo has also helped in no small way. I really appreciate his effort rendered to me during the course of this study.

My profound gratitude goes to Mallam Ibrahim Mohammed Bomoi (Auditor General for MFCT Area Council) your kindness is acknowledged.

I must recognize and acknowledge the support I received from my immediate family with every sense of sincerity. I deeply appreciate all they did for me by way of moral and spiritual support. I appreciate my Little Ibrahim for his sacrifice and support to make sure I succeed in my endeavor.

Furthermore, I am warmly indebted to Dr. and Mrs. Babangida Aliyu, Mr. and Mrs. M. K. Ibrahim, Mr and mrs. Sa'id; your concern is highly appreciated.

I owe much thanks to my good course mates, for their contributions in their own different ways.

TABLE OF CONTENTS

CHAPTER ONE: INTRODUCTION

CHAPTER TWO: LITERATURE REVIEW

CHAPTER THREE: SYSTEM ANALYSIS AND DESIGN

CHAPTER FOUR: SOFTWARE DESIGN AND IMPLEMENTATION

CHAPTER FIVE: SUMMARY/RECOMMENDATION

CHAPTER ONE

INTRODUCTION

1.1 A History of Java

Perhaps the microprocessor revolution's most important contribution to date is that it made possible the development of personal computers which may soon number 300 million world-wide. Personal computers have had a profound impact on people and the way organisations conduct and manage their business.

Many people believe that the next major area in which microprocessors will have a, profound impact is in intelligent consumer electronic devices. Recognising this, Sun Microsystems funded an internal corporate research project code-named Green in 1991. The project resulted in the development of a C and C++ based language which its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee place, the name ***Java*** was suggested and it stuck.

But the Green project ran into some difficulties. The marketplace for intelligent consumer electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. So the project was in danger of being cancelled. By sheer good fortune, the World Wide Web exploded in popularity in 1993 and Sun people saw the immediate potential of using Java to create Web pages with so-called dynamic content. This breathed new life into the project.

Sun formally announced Java at a major conference in May 1995. Ordinarily, an event like this would not have generated much attention. However, Java generated immediate interest in the business Community because of the phenomenal interest in the World Wide Web. Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of

World Wide Web servers (the computers that provide the content we see in our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants), and so on.

1.2 Java Class Libraries

Java programs consist of pieces called classes. Classes consist of pieces called methods that perform tasks and return information when they complete their tasks. You can program each piece you may need to form a Java program. But most Java programmers take advantage of rich collections of existing classes in Java class libraries. The class libraries are also known as the Java APIs (Applications Programming Interfaces). Thus, there are really two pieces to learning the Java "world". The first is learning the Java language itself so that you can program your own classes and the second is learning how to use the classes in the extensive Java class libraries. Class libraries are provided primarily by compiler vendors; but many class libraries are supplied by independent software vendors. Also, many class libraries are available from the Internet and World Wide Web as shareware (products you can download for a small fee) and freeware (products you can download for free).

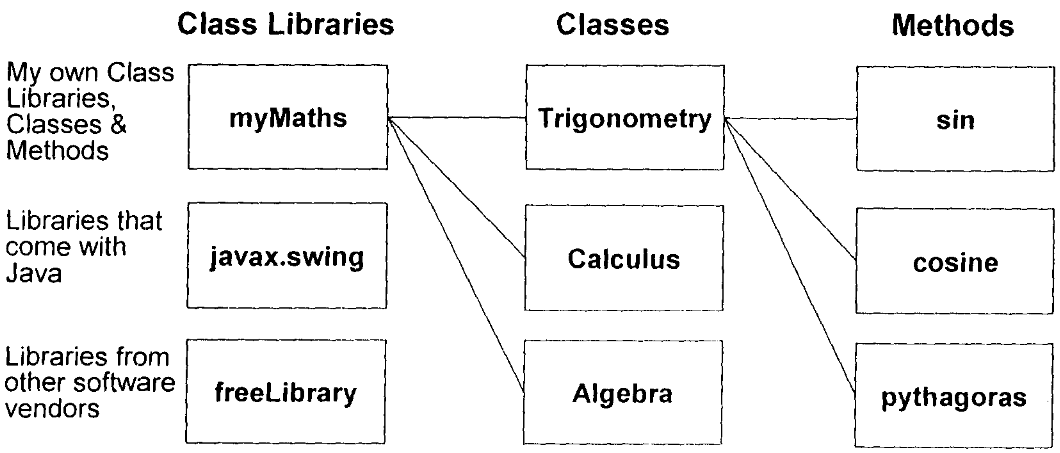

Class Libraries
Classes
Methods
My own Class Libraries, Classes & Methods
myMaths
Trigonometry
sin
Libraries that come with Java
javax.swing
Calculus
cosine
Libraries from other software vendors
freeLibrary
Algebra
pythagoras

Figure 1.2 Class Libraries, Classes and Methods.

The advantage of creating your own classes and methods is that you will know exactly how they work. You will be able to examine the Java code. The disadvantage is the time-consuming and complex effort that goes into designing and developing new classes and methods.

1.3 Basics of a Typical Java Environment

Java systems generally consist of several parts: an environment, the language, the Java Applications Programming Interface (API), and various class libraries. The following discussion explains a typical Java program development environment as shown below.

Java programs normally go through five phases to be executed (Figure 1.1). These are: **edit**, **compile**, **load**, **verify** and **execute**.

Phase 1 consists of editing a file. This is accomplished with an editor program. The programmer types a Java program using the editor and makes corrections if necessary. When the programmer specifies that the file in the editor should be saved, the program is stored on a secondary storage device such as a disk. Java program file names end with the .java extension. Java integrated development environments (IDEs) such as Netbeans, Eclipse, and a host of others, have built-in editors that are smoothly integrated into the programming environment.

In **Phase 2**, the programmer gives the command **javac** to *compile* the program. The Java compiler translates the Java program into bytecodes the language understood by the Java interpreter

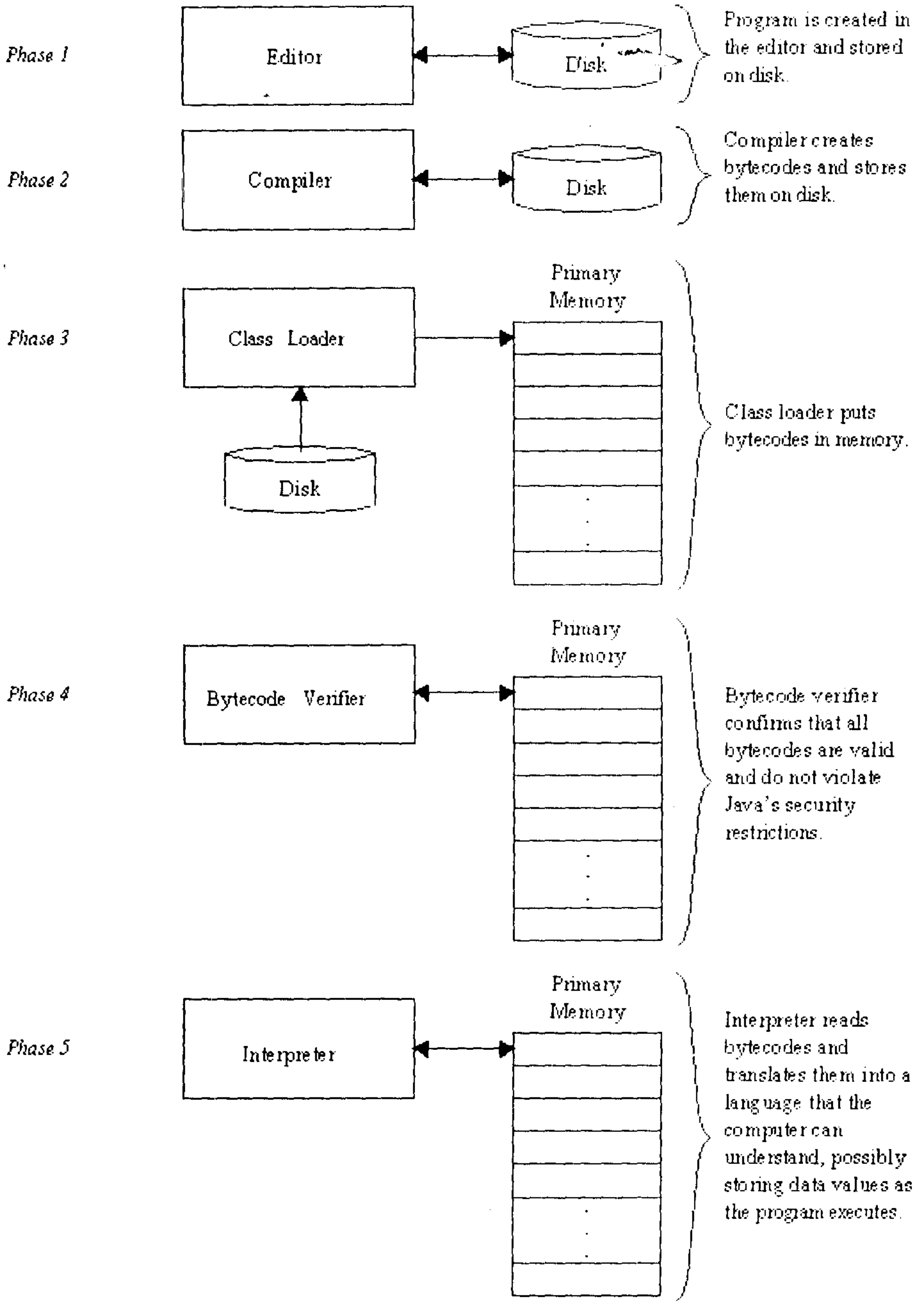

Phase 1
Editor
Disk
Program is created in the editor and stored on disk.
Phase 2
Compiler
Disk
Compiler creates bytecodes and stores them on disk.
Phase 3
Class Loader
Disk
Primary Memory
Class loader puts bytecodes in memory.
Phase 4
Bytecode Verifier
Primary Memory
Bytecode verifier confirms that all bytecodes are valid and do not violate Java's security restrictions.
Phase 5
Interpreter
Primary Memory
Interpreter reads bytecodes and translates them into a language that the computer can understand, possibly storing data values as the program executes.

Figure 1.1 – A typical Java environment.

Phase 3 is called *loading*. The program must first be placed in memory before it can be executed. This is done by the class loader, which takes the class file (or files) containing the bytecodes and transfers it to memory. The .class file can be loaded from a disk on your system or over a network (such as your local university or company network or even the Internet). There are two types of programs for which the class loader loads **.class** files – applications and applets. An application is a program such as a word processor program, a spreadsheet program, a drawing program, an email program, etc. that is normally stored and executed from the user's local computer. An applet is a small program that is normally stored on a remote computer that users connect to via a World Wide Web browser. Applets are loaded from a remote computer into the browser, executed in the browser and discarded when execution completes. To execute an applet again, the user must point their browser at the appropriate location on the World Wide Web and reload the program into the browser. Applications are loaded into memory and executed using the Java interpreter via the command **java**.

The class loader also is executed when a Java applet is loaded into a World Wide Web browser such as Netscape's Communicator or Microsoft's Internet Explorer. Browsers are used to view documents on the World Wide Web called HTML (Hypertext Mark-up Language) documents. HTML is used to format a document in a manner that is easily understood by the browser application. An HTML document may refer to a Java applet. When the browser sees an applet referenced in an HTML document, the browser launches the Java class loader to load the applet (normally from the location where the HTML document is stored). Browsers that support Java each have a built-in Java interpreter. Once the applet is loaded, the browser's Java interpreter executes the applet.

Before the bytecodes in an applet are executed by the Java interpreter built into a browser, they are verified by the *bytecode verifier* in Phase 4 (this also happens in applications that download bytecodes from a network). This ensures that the bytecodes for classes that are loaded from the Internet (referred to as downloaded classes) are valid and that they do not violate Java's security restrictions. Java enforces strong security because Java programs arriving over the network should not be able to cause damage to your files and your system (as computer viruses might).

Finally, in **Phase 5**, the computer, under the control of its CPU, interprets the program one bytecode at a time, thus performing the actions specified by the program. Programs may not work on the first try. Each of the preceding phases can fail because of various errors that we will discuss in this text. For example, an executing program might attempt to divide by zero (an illegal operation in Java just as it is in arithmetic). This would cause the Java program to print an error message. The programmer would return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine if the corrections work properly.

Most programs in Java input and/or output data. When we say that a program prints a result, we normally mean that the result is displayed on a screen. Data may be output to other devices such as disks and hardcopy printers.

Perhaps the most striking problem with early versions of Java is that Java programs execute interpretively on the clients machine. Interpreters execute slowly compared to fully compiled machine code. *Interpreters have an advantage over compilers for the Java world, namely that an interpreted program can begin execution immediately as soon as it is downloaded to the client's machine, whereas a source program to be compiled must first suffer a potentially long delay as the program is compiled before it can be executed.*

Although only Java interpreters were available to execute bytecodes at the client's site on early Java systems, Java compilers have been written for most popular platforms. These compilers take the Java bytecodes (or in some cases the Java source code) and compile them into the native machine code of the client's machine. These compiled programs perform comparably to compiled C or C++ code. Because there are compilers for every Java platform, Java programs will perform at the same level on all platforms.

An intermediate step between interpreters and compilers is a just-in-time (JIT) compiler that, as the interpreter runs, produces compiled code for the programs and executes the programs in machine language rather than reinterpreting them. JIT compilers do not produce machine language that is as efficient as a full compiler. Full compilers for Java are under development now.

1.4 Justification for the Study

The E-Yearbook uses Oracle 10g Express Edition to store address information such as names, pictures, phone numbers, email addresses, personal statements and postal addresses. It allows you to create new address entries and to save, edit, and delete them. The application creates its tables the first time if they don't already exist, a task that can also be done from the Oracle 10g XE database Home page. The application uses a Data Access Object (DAO) to isolate the database-specific code. The DAO encapsulates database connections and statements. A DAO is a useful design pattern that allows loose coupling between an application and the underlying persistence-storage mechanism.

1.5 Aim and objective of project

Every effective system gains its quality from the speed at which it executes its dedicated task, which is a subject of time and other resources required such as an effective information system. Therefore this project is aimed at

- Developing a Java Application, with an oracle XE Database backend, that manages students' personal information for the purpose of sharing this with other students after graduation.

CHAPTER TWO

LITERATURE REVIEW

2.1 Java's Past, Present, and Future

Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations. Modeled after C++, the Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level (more about this later). Java is often mentioned in the same breath as HotJava, a World Wide Web browser from Sun like Netscape or Mosaic. What makes HotJava different from most other browsers is that, in addition to all its basic Web features, it can also download and play applets on the reader's system. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive. Applets can be used to create animations, figures, or areas that can respond to input from the reader, games, or other interactive effects on the same Web pages among the text and graphics. Although HotJava was the first World Wide Web browser to be able to play Java applets, Java support is rapidly becoming available in other browsers. Netscape provides support for Java applets, and other browser developers have also announced support for Java in forthcoming products. Java's goals at that time were to be small, fast, efficient, and easily portable to a wide range of hardware devices. It is those same goals that made Java an ideal language for distributing executable programs via the World Wide Web, and also a general-purpose programming language for developing programs that are easily usable and portable across different platforms.

The Java language was used in several projects within Sun, but did not get very much commercial attention until it was paired with HotJava. HotJava was written in 1994 in a matter of months, both as a vehicle for downloading and running applets and also as ân example of the sort of complex application that can be written in Java. Presently, Sun has released the 6th version of the Java Developer's Kit (JDK1.6.0), which includes tools for developing Java applets and applications on Sun systems running Solaris, for Windows NT and Linux.

The JDK does include an application called ***appletviewer*** that allows you to test your Java applets as you write them. If an applet works in the appletviewer, it should work with any Java-capable browser.

2.2 Java Is Platform-Independent

Platform independence is one of the most significant advantages that Java has over other programming languages, particularly for systems that need to work on many different platforms. Java is platform-independent at both the source and the binary level. ***Platform-independence*** is a program's capability of moving easily from one computer system to another.

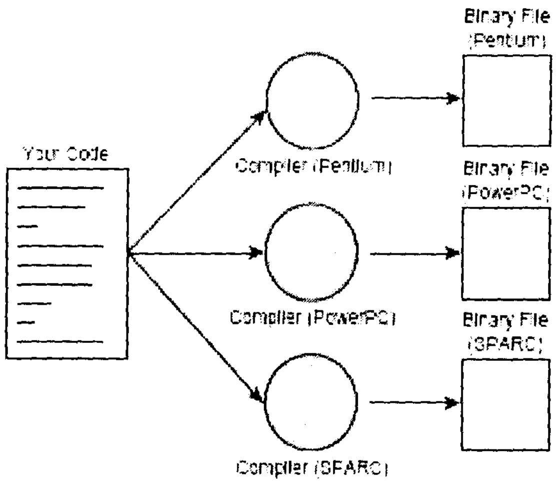
At the source level, Java's primitive data types have consistent sizes across all development platforms. Java's foundation class libraries make it easy to write code that can be moved from platform to platform without the need to rewrite it to work with that platform.

Platform-independence doesn't stop at the source level, however. Java binary files are also platform-independent and can run on multiple problems without the need to recompile the source. How does this work? Java binary files are actually in a form called bytecodes. ***Bytecodes*** are a set of instructions that looks a lot like some machine codes, but that is not specific to any one processor.

Normally, when you compile a program written in C or in most other languages, the compiler translates your program into machine codes or processor instructions. Those instructions are specific to the processor your computer is running—so, for example, if you compile your code on a Pentium system, the resulting program will run only on other Pentium systems. If you want to use the same program on another system, you have to go back to your original source, get a compiler for that system, and recompile your code. Figure 1.2 shows the result of this system: multiple executable programs for multiple systems.

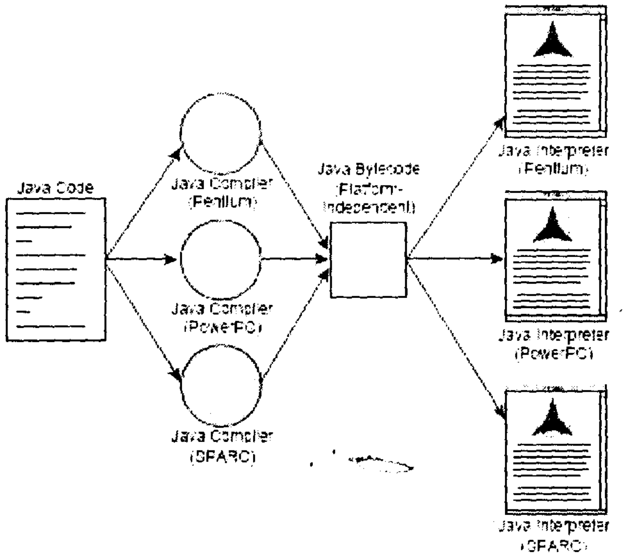
Things are different when you write code in Java. The Java development environment has two parts: a Java compiler and a Java interpreter. The Java compiler takes your Java program and instead of generating machine codes from your source files, it generates bytecodes.

Figure 1.2.
Traditional compiled programs.



To run a Java program, you run a program called a bytecode interpreter, which in turn executes your Java program (see Figure 1.3). You can either run the interpreter by itself, or — for applets — there is a bytecode interpreter built into HotJava and other Java-capable browsers that runs the applet for you.

Figure 1.3.
Java programs.



Why go through all the trouble of adding this extra layer of the bytecode interpreter? Having your Java programs in bytecode form means that instead of being specific to any one system, your programs can be run on any platform and any operating or window system as long as the Java interpreter is available. This capability of a single binary file to be executable across platforms is crucial to what enables applets to work, because the World Wide Web itself is also platform-independent. Just as HTML files can be read on any platform, so applets can be executed on any platform that is a Java-capable browser.

2.3 Object-Oriented programming

To some, object-oriented programming (OOP) technique is merely a way of organizing programs, and it can be accomplished using any language. Working with a real object-oriented language and programming environment, however, enables you to take full advantage of object oriented methodology and its capabilities of creating flexible, modular programs and reusing code.

Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions. Because these class libraries are written in Java, they are portable across platforms as all Java applications are.

Java Is Easy to Learn

In addition to its portability and object-orientation, one of Java's initial design goals was to be small and simple, and therefore easier to write, easier to compile, easier to debug, and, best of all, easy to learn. Keeping the language small also makes it more robust because there are fewer chances for programmers to make difficult-to-find mistakes. Despite its size and simple design, however, Java still has a great deal of power and flexibility.

Java is modeled after C and C++, and much of the syntax and object-oriented structure is borrowed from the latter. If you are familiar with C++, learning Java will be particularly easy for you, because you have most of the foundation already.

Although Java looks similar to C and C++, most of the more complex parts of those languages have been excluded from Java, making the language simpler without sacrificing much of its power. There are no pointers in Java, nor is there pointer arithmetic. Strings and arrays are real objects in Java. Memory management is automatic. To an experienced programmer, these omissions may be difficult to get used to, but to beginners or programmers who have worked in other languages, they make the Java language far easier to learn.

Object-oriented programming is modeled on how, in the real world, objects are often made up of many kinds of smaller objects. This capability of combining objects, however, is only one very general aspect of object-oriented programming.

Object-oriented programming provides several other concepts and features to make creating and using objects easier and more flexible, and the most important of these features is that of classes. A ***class*** is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects. When you write a program in an object-oriented language, you don't define actual objects. You define classes of objects.

For example, you might have a Tree class that describes the features of all trees (has leaves and roots, grows, creates chlorophyll). The Tree class serves as an abstract model for the concept of a tree—to reach out and grab, or interact with, or cut down a tree you have to have a concrete instance of that tree. Of course, once you have a tree class, you can create lots of different instances of that tree, and each different tree instance can have different features (short, tall, bushy, drops leaves in autumn), while still behaving like and being immediately recognizable as a tree.

An ***instance*** of a class is another word for an actual object. If classes are an abstract representation of an object, an instance is its concrete representation. So what, precisely, is the difference between an instance and an object? Nothing, really. Object is the more general term, but both instances and objects are the concrete representation of a class. In fact, the terms instance and object are often used interchangeably in OOP language. An instance of a tree and a tree object are both the same thing. When you write a Java program, you design and construct a set of classes. Then, when your program runs, instances of those classes are created and discarded as needed. Your task, as a Java programmer, is to create the right set of classes to accomplish what your program needs to accomplish. Fortunately, you don't have to start from the very beginning: the Java environment comes with a library of classes that implement a lot of the basic behavior you need—not only for basic programming tasks (classes to provide basic math functions, arrays, strings, and so on), but also for graphics and networking behavior. In many cases, the Java class libraries may be enough so that all you have to do in your Java program is create a single class that uses the standard class libraries. For complicated Java programs, you may have to create a whole set of classes with defined interactions between them. A ***class library*** is a set of classes.

2.4 Relational Databases

A ***database*** is a structured collection of meaningful information stored over a period of time in machine-readable form for subsequent retrieval. This definition is fairly intuitive and says nothing about structure or methodology. By this definition, any file or collection of files can be considered a database. However, to be useful in practical terms, a database must form part of a system that provides for the management of the data it contains. Seen from this perspective, a database must be more than a mere collection of files. It must be a complete system.

A ***practical* database** management system combines the physical storage of data with the capability to manage and interact with the data. Such a system must support the following tasks:

- Creation and management of a logical data structure

- . Data entry and retrieval
- . Manipulation of the data in a logical and consistent manner
- Storage of data reliably over a significant period of time

Prior to the development of modern relational databases, a number of different approaches were tried. In many cases, these were simple, proprietary data-storage systems designed around a specific application. However, large corporations, notably IBM, were marketing more general solutions.

2.3.1 The Relational Model

The big step forward in database technology was the development of the ***relational database model***. The relational database derives from work done in the late 1960s by E.F. Codd, a mathematician at IBM. His model is based on the mathematics of set theory and predicate logic. In fact, the term ***relational*** has its roots in the mathematical terminology of Codd's paper entitled "A relational model of data for large shared data banks," which was published in *Communications of the ACM*, Vol. 13, No. 6, June 1970, pp. 377-387. In this paper, Codd uses the terms *relation*, *attribute*, and *tuple* where more common programming usage refers to *table*, *column*, and *row*, respectively.

The importance of Codd's ideas is such that the term "database" generally refers to a relational database. Similarly, in common usage, a Database Management System, or DBMS, generally means a Relational Database Management System. The terms are always used interchangeably.

Codd's model covers the three primary requirements of a relational database: structure, integrity, and data manipulation. The fundamentals of the relational model are as follows:

- A relational database consists of a number of unordered tables.
- The structure of these tables is independent of the physical storage medium used to store the data.
- The contents of the tables can be manipulated using nonprocedural operations that return tables.

The implementation of Codd's relational model means that a user does not need to understand the physical structure of the data in order to access and manage data in the database. Rather than accessing data by referring to files or using pointers, the user accesses data through a common tabular architecture. The relational model maintains a clear distinction between the logical views of the data presented to the user and the physical structure of the data stored in the system.

Codd based his model on a simple tabular structure, though his term for a table was a *relation*. Each table is made up of one or more rows (or *tuples*). Each row contains a number of fields, corresponding to the columns or *attributes* of the table.

The tabular structure Codd defines is simple and relatively easy for the user to understand. It is also sufficiently general to be capable of representing most types of data in virtually any kind of structure. An additional advantage of a tabular structure is that tables are amenable to manipulation by a clearly defined set of mathematical operations that generate results that are also in the form of tables. These mathematical operations lend themselves readily to implementation in a high-level language. In fact, Codd's rules require that a high level language be incorporated in the RDBMS for just this purpose. That language has evolved into the Structured Query Language, SQL.

The use of a high-level language to manipulate the data at the logical level is an important feature, providing a level of abstraction which lets the user insert or retrieve data from the tables based on attributes of the data rather than its physical structure. For example, rather than requiring the user to retrieve a number stored in a certain location on disk, the use of a high-level query language allows the user to request the checking balance of a particular customer's account by account number or customer name.

A further advantage of this approach is that, while the user defines his or her requests in logical terms, the database management system (DBMS) can implement them in a highly optimized manner with respect to the physical implementation of the storage system. By decoupling the logical operations from the physical operations, the DBMS can achieve a combination of user friendliness and efficiency that would not otherwise be possible.

2.4.2 Codd's Rules

When Codd initially presented his paper, the meaning of the relational model he described was not widely understood. To clarify his ideas, Codd published his famous Fidelity Rules. In theory, a RDBMS must conform to these rules. As it turns out, some of these rules are extremely difficult to implement in practice, so no existing RDBMS complies fully.

For example, Rule 1, the Information Rule, requires that all data be represented as values in tables; it is important to understand the idea of tables before moving on to discuss Rule 0, which requires that the database be managed in accordance with its own rules for managing data.

Tables, Rows, Columns, and Keys

Codd's Information Rule (Rule 1) states that all data in a relational database must be explicitly represented at the logical level as values in tables and in no other way. In other words, tables are the basis of any RDBMS. *Tables* in the relational model are used to represent collections of objects or events in the real world. A single table should represent a collection of a single type of object, such as customers or inventory items.

All relational databases rely on the following design concepts:

- All data in a relational database is explicitly represented at the logical level as values in tables.
- Each cell of a table contains the value of a single data item.
- Cells in the same column are members of a set of similar items.
- Cells in the same row are members of a group of related items.
- Each table defines a key made up of one or more columns that uniquely identify each row.

Nulls

In a practical database, situations arise in which you either don't know the value of a data element or don't have an applicable value. Does that blow away the whole table? The answer lies in the concept of ***systematic nulls***.

Codd's Systematic Nulls Rule (Rule 3) states that the RDBMS is required to support a representation of missing and inapplicable information that is systematic, distinct from all regular values, and independent of data type. In other words, a relational database must allow the user to insert a NULL when the value for a field is unknown or not applicable. Clearly, the requirement to support NULLS means that the RDBMS must be able to handle NULL values in the course of normal operations in a systematic way. This is managed through the ability to insert, retrieve, and test for NULLS and to specify NULLS as valid or invalid column va lues.

Primary Keys

Codd's Guaranteed Access Rule (Rule 2) states that every data element must be logically accessible through the use of a combination of its primary key name, primary key value, table name, and column name. This is guaranteed by designating a ***primary key*** that contains a unique value for each row in the table. Each table can have only one primary key, which can be any column or group of columns in the table having a unique value for each row.

It is worth noting that, while most relational database management systems will let you create a table without a primary key, the usability of the table will be compromised if you fail to assign a primary key. The reason for this is that one of the strengths of a relational database is the ability to link tables to each other. These links between tables rely on using the primary key as a linking mechanism.

Primary keys can be *simple* or *composite*. A simple key is a key made up of one column, whereas a composite key is made up of two or more columns. Although there is no absolute rule as to how you select a column or group of columns for use as a primary key, the decision should usually be based upon common sense. In other words, you should base your choice of a primary key upon the following factors:

- Use the smallest number columns necessary, to make key access efficient.
- Use columns or groups of columns that are unlikely to change, since changes will break links between tables.
- Use columns or groups of columns that are both simple and understandable to users.

In practice, the most common type of key is a column of unique integers specifically created for use as the primary key. The unique integer serves as a row identifier or ID for each row in the table. Oracle, in fact, defines a special ROW_ID pseudo column and Access has an AutoNumber data type commonly used for this purpose.

Another good reason to use a unique integer as a primary key is that integer comparisons are far more efficient than string comparisons. This means that accessing data using a single integer as a key is faster than using a string or, in the case of a multiple column key, several integers or strings.

Note: Since primary keys are used as unique row identifiers, they can never have a NULL value. The NOT NULL integrity constraint must be applied to a column designated as a primary key. Many Relational database Management Systems apply the NOT NULL constraint to primary keys automatically.

Foreign Keys

A ***foreign key*** is a column in a table used to refe rence a primary key in another table. If your database contains only one table, or a number of unrelated tables, you won't have much use for your primary key. The primary key becomes important when you need to work with multiple tables. For example, in the tables used in our application we have three tables: Addresses, Blob_content and Clob_content.

2.4.3 Relationships

As illustrated in the preceding discussions of primary and foreign keys, they are defined to model the relationships among the different tables in a database. These tables can be related in one of three ways:

- One-to-one

- One-to-many
- Many-to-many

One-to-one relationships

In a *one-to-one* relationship, every row in the first table has a corresponding row in the second table. This type of relationship is often created to separate different types of data for security reasons. For example, you might want to keep confidential information such as credit-card data separate from less restricted information.

Another common reason for creating tables with a one -to-one relationship is to simplify implementation. For example, if you are creating a Web application involving several forms, you might want to use a separate table for each form.

Other reasons for breaking a table into smaller parts with one-to-one relationships between them are to improve performance or to overcome inherent restrictions such as the maximum column count that a database system supports. Tables related in a one-to-one relationship should always have the same primary key. This is used to perform joins when the related tables are queried together.

One-to-many relationships

In a *one-to-many* relationship, every row in the first table can have zero, one, or many corresponding rows in the second table. But for every row in the second table, there is exactly one row in the first table. One-to-many relationships are also sometimes called *parent-child* or *master-detail* relationships because they are commonly used for lookup tables.

Many-to-many relationships

In a *many-to-many* relationship, every row in the first table can have many corresponding rows in the second table, and every row in the second table can have many corresponding rows in the first table. Many-to-many relationships can't be directly modeled in a relational database. They must be broken into multiple one-to-many relationships.

Normalization

Normalization is the process of organizing the data in a database by making it conform to a set of rules known as the normal forms. The *normal forms* are a set of design guidelines that are designed to eliminate redundancies and to ensure consistent dependencies. Apart from wasting space, redundant data creates maintenance problems. For example, if you save a customer's address in two locations, you need to be absolutely certain to make any required changes in both locations.

It is important to ensure that data dependencies are consistent so that you can access and manipulate data in a logical and consistent manner. Although normalization enhances the integrity of the data by minimizing redundancy and inconsistency, it does so at the cost of some impact on performance.

Data-retrieval efficiency can be reduced, since applying the normalization rules can result in data being redistributed across multiple records. A database that conforms to the normalization rules is said to be in ***normal form***. If the database conforms to the first rule, the database is said to be in ***first normal form,*** abbreviated as ***1NF***. If it conforms to the first four rules, the database is considered to be in ***fourth normal form (4NF)***.

First normal form

The requirements of the first normal form are as follows:

- All records have the same number of fields.
- All fields contain only a single data item.
- There must be no repeated fields.

The first of these requirements, that all occurrences of a record type must contain the same number of fields, is a built-in feature of all database systems.

The second requirement, that all fields contain only one data item, ensures that you can retrieve data items individually. This requirement is also known as the *atomicity* requirement. Requiring that each data item be stored in only one field in a record is important to ensure data integrity.

Finally, each row in the table must be identified using a unique column or set of columns. This unique identifier is the primary key.

Second normal form

The requirements of the second normal form are as follows:

- The table must be in first normal form.
- The table cannot contain fields that do not contain information related to the whole of the key.

The second normal form is only relevant when a table has a multipart key. Second normal form requires that a table should only contain data related to one entity, and that entity should be described by its primary key. In addition, of course, storing the same data item in multiple locations is very inefficient in terms of space, and requires that any change to the data item be made to all rows containing the data item, rather than to a single master reference.

Third normal form

The requirements of the third normal form are as follows:

- The table must be in second normal form.
- The table cannot contain fields that are not related to the primary key.

Third normal form is very similar to second normal form, with the exception that it covers situations involving simple keys rather than compound keys.

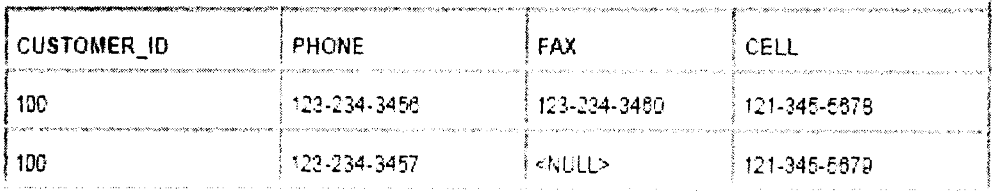
Fourth normal form

The requirements of the fourth normal form are as follows:

- The table must be in third normal form.
- The table cannot contain two or more independent multivalued facts about an entity.

For example, if you wanted to keep track of customer phone numbers, you could create a new table containing a Customer_ID number column, a phone number column, a fax number column, and a cell-phone number column. As long as a customer has only one of each listed in the table, there is no problem. However, if a customer has two land line phones, a fax, and two cell phones, you might be tempted to enter the numbers as shown in Table 1-14.

Table 1-14: Phone Numbers Table which violates 4NF

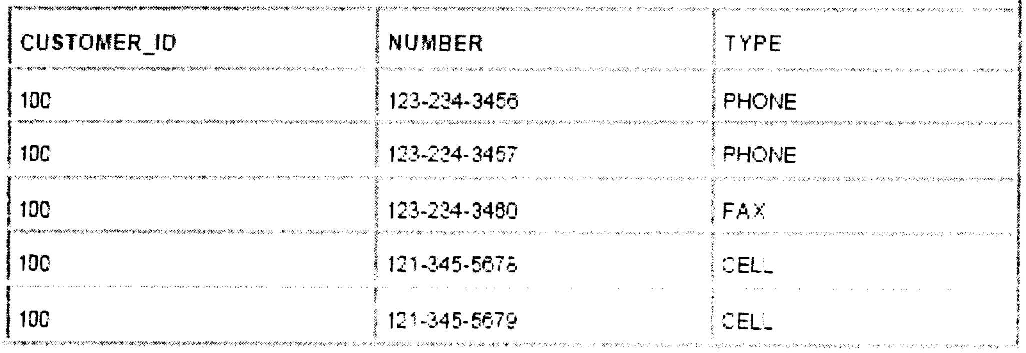


CUSTOMER_ID	PHONE	FAX	CELL
100	123-234-3456	123-234-3460	121-345-6878
100	123-234-3457	<NULL>	121-345-6879

Since there is no relationship between the different phone numbers in a given row, this table violates the fourth normal form, in that there are two or more independent multivalued facts (or phone numbers) for the customer on each row. The *combinations* of land line, fax, and cell phone numbers on a given row are not meaningful. The main problem with violating the fourth normal form is that there is no obvious way to maintain the data. If, for example, the customer decides to give up the cell phone listed in the first row, should the cell p hone number in the second row be moved to the first row, or left where it is? If he or she gives up the land line phone in the second row and the cell phone in the first row, should all the phone numbers be consolidated into one row? Clearly, the maintenance of this database could become very complicated.

The solution is to design around this problem by deleting the phone, fax, and cell columns from the original table and creating an additional table containing Customer_ID as a foreign key, and phone number and type as data fields (see Table 1-15). This will allow you to handle several phone numbers of different types for each customer without violating the fourth normal form.

Table 1-15: Phone Numbers Table



CUSTOMER_ID	NUMBER	TYPE
100	123-234-3456	PHONE
100	123-234-3457	PHONE
100	123-234-3460	FAX
100	121-345-5678	CELL
100	121-345-5679	CELL

Fifth normal form

The requirements of the fifth normal form are as follows:

- The table must be in fourth normal form.
- It must be impossible to break down a table into smaller tables unless those tables logically have the same primary key as the original table.

The fifth normal form is similar to the fourth normal form, except that where the fourth normal form deals with independent multivalued facts, the fifth normal form deals with interdependent multivalued facts.

2.5 Structured Query Language (SQL)

The Structured Query Language (SQL) was first developed by IBM in the 1970s and was later the subject of several ANSI standards. As a result of the way that the requirements for a high-level database language are defined, SQL is usually considered to be composed of a number of sublanguages. These sublanguages are as follows: Data Definition Language (DDL) is used to create, alter, and drop tables and indexes.

- Data Manipulation Language (DML) is used to insert, update, and delete data.
- Data Query Language (DQL) is used to query the database using the SELECT command.
- Transaction Control Commands are used to start, commit, or rollback transactions.
- Data Control Language (DCL) is used to grant and revoke user privileges and to change passwords.

Despite the conventional division of SQL into a number of sublanguages, statements from any of these constituent sublanguages can be used together. The convention is really just a reflection of the way Codd's rules define the requirement for a high level language, with sublanguages for different functions.

Data Definition Language

Data definition operations are handled by SQL's Data Definition Language, which is used to create and modify a database. The SQL2 standard refers to DDL statements as "SQL Schema Statements." The SQL standard defines a Schema as a high level abstraction of a container object which contains other database objects.

A good example of the use of the DDL is the creation of a table.

Data Manipulation Language

The Data Manipulation Language comprises the SQL commands used to insert data into a table and to update or delete data. SQL provides the following three statements you can use to manipulate data within a database:

- INSERT
- UPDATE
- DELETE

The INSERT statement is used to insert data into a table, one row or record at a time. It can also be used in combination with a SELECT statement to perform bulk inserts of multiple selected rows from another table or tables.

The UPDATE command is used to modify the contents of individual columns within a set of rows. The UPDATE command is normally used with a WHERE clause, which is used to select the rows to be updated.

The DELETE command is used to delete selected rows from a table. Again, row selection is based o n the result of an optional WHERE clause.

Data Query Language

The Data Query Language is the portion of SQL used to retrieve data from a database in response to a query. The SELECT statement is the heart of a SQL query. In addition to its use in returning data in a query, it can be used in combination with other SQL commands to select data for a variety of other operations, such as modifying specific records using the UPDATE command.

The most common way to use SELECT, however, is as the basis of data retrieval commands, or queries, to the database. The basic form of a simple query specifies the names of the columns to be returned and the name of the table they can be found in. A basic SELECT command looks like this:

```
SELECT columnName1, columnName2,.. FROM tableName;
```

In addition to this specific form, where the names of all the fields you want returned are specified in the query, SQL supports a wild-card form. In the wild-card form, an asterisk (*) is substituted for the column list, as shown here:

```
SELECT * FROM tableName;
```

The wild card tells the database management system to return the values for all columns.

The real power of the SELECT command comes from the use of the WHERE clause. The WHERE clause allows you to restrict the query to return the requested fields from only records that match some specific criteria. For example, you can query the Customers Table:

```
SELECT * FROM Contact_Info WHERE Last_Name = 'Corleone';
```

2.6 Transaction Management and the Transaction Control Commands

Transaction management refers to the capability of an RDBMS to execute database commands in groups, known as *transactions* . A transaction is a group or sequence of commands, all of which must be executed in order and all of which must completed successfully. The Transaction Control Commands are used to control transactions.

The ACID Test

A commonly used expression in data processing is the ACID test. The ACID test defines a set of properties that a database management system must have in order to be adequate for handling transactions. These properties are as follows:

- Atomicity
- Consistency
- Isolation
- Durability

A discussion of the preceding properties follows.

Atomicity

Transactions must be *atomic*. Specifically, a transaction must be executed in its entirety and committed as a whole or rolled back as a whole, so that either all changes that constitute a transaction take effect or none of them take effect. A classic example of an atomic transaction is a transfer of funds from a checking account to a savings account. Clearly, you want both the deduction from savings and the addition to checking to take place, failing which, neither should take place. When atomicity is not guaranteed, you have an accounting nightmare.

Consistency

The *consistency* requirement defines a transaction as *legal* only if it obeys user-defined integrity constraints. Essentially, these constraints define legal database states and proscribe transactions that cause transitions from a legal state to an illegal state. For example, if you are making a transfer of funds from a checking account to a savings account and your business rules require that such a transfer be logged to another table, any problems updating that table will violate the integrity constraint and will require that the entire transaction be rolled back.

Isolation

Isolation means that the effects of a transaction must be invisible to other transactions until the current transaction is complete. For example, if you are making a transfer of funds from a checking account to a savings account, the intermediate balances after savings have been debited, but before checking has been credited, must not be available to an outside transaction. If the intermediate balances are available to an outside transaction, you might, for example, generate an insufficient funds warning, since the funds will show up in neither account.

Durability

The *durability* requirement demands that, once committed, the results of a transaction be preserved in some form of long term storage. In other words, once a funds transfer has been made from savings to checking, the DBMS must save it to persistent storage.

Transaction Management in SQL

If anything goes wrong during the transaction, the database management system allows the entire transaction to be cancelled, or "Rolled Back." If, on the other hand, it completes successfully, the transaction can be saved to the database, or "Committed." A transaction typically involves several related commands, as in the case of a bank transfer. If a client orders a transfer of funds from his savings account to his checking account, at least these two database-access commands must be executed:

- The savings account must be debited.
- The checking account must be credited.

If one of these commands is executed and the other is not, the funds will either vanish from the savings account without appearing in the checking account, or the funds will be credited to the checking account without being withdrawn from the savings account. The solution is to combine logically related commands into groups that are committed as a single transaction. If a problem arises, the entire transaction can be rolled back, and the problem can be fixed without serious adverse impact on business operations.

SQL supports this requirement through the COMMIT and ROLLBACK commands. The COMMIT command commits changes made from the beginning of the transaction to the point at which the command is issued, and the ROLLBACK command undoes them. In addition, most databases support the AUTOCOMMIT option, which tells the database management system to commit all commands individually as they are executed. This option can be turned on or off with the SET command. By default, the AUTOCOMMIT option is usually on.

CHAPTER THREE

SYSTEM ANALYSIS AND DESIGN

3.1 E-Yearbook Application

The E-Yearbook application uses Oracle XE Database to store address information. It stores names, phone numbers, email addresses, pictures etc. It allows you to create new address entries and to save, edit, and delete them. Oracle 10g Express Edition must be installed for the application to work. To deploy this database application, we need only the application JAR file and the Oracle database **jdbc** library JAR file.

JDBC is a Java Database Connectivity API that lets you access virtually any tabular data source from a Java application. In addition to providing connectivity to a wide range of SQL databases, JDBC allows you to access other tabular data sources such as spreadsheets or flat files. Although JDBC is often thought of as an acronym for *Java Database Connectivity*, the trademarked API name is actually JDBC.

JDBC defines a low-level API designed to support basic SQL functionality independently of any specific SQL implementation. This means the focus is on executing raw SQL statements and retrieving their results. JDBC is based on the X/Open SQL Call Level Interface, an international standard for programming access to SQL databases, which is also the basis for Microsoft's ODBC interface.

The JDBC 2.0 API includes two packages: java.sql, known as the JDBC 2.0 core API; and javax.sql, known as the JDBC Standard Extension. Together, they contain the necessary classes to develop database applications using Java. As a core of the Java 2 Platform, the JDBC is available on any platform running Java.

The JDBC 3.0 Specification, released in October 2001, introduces several features, including extensions to the support of various data types, additional MetaData capabilities, and enhancements to a number of interfaces.

The JDBC Extension Package (javax.sql) was introduced to contain the parts of the JDBC API that are closely related to other pieces of the Java platform that are themselves optional packages, such as the Java Naming and Directory Interface (JNDI) and the Java Transaction Service (JTS). In addition, some advanced features that are easily separable from the core JDBC API, such as connection pooling and rowsets, have been added to javax.sql. Putting these advanced facilities into an optional package instead of into the JDBC 2.0 core API helps to keep the core JDBC API small and focused.

The main strength of JDBC is that it is designed to work in exactly the same way with any relational database. In other words, it isn't necessary to write one program to access an Oracle database, another to access a Sybase database, another for SQL Server, and so on. JDBC provides a uniform interface on top of a variety of different database-connectivity modules. As we have demonstrated in this application, a single program written using JDBC can be used to create a SQL interface to virtually any relational database. The three main functions of JDBC are as follows:

- Establishing a connection with a database or other tabular data source
- Sending SQL commands to the database
- Processing the results

The JDBC API defines standard mappings between SQL data types and Java/JDBC data types, including support for Oracle 10g advanced data types such as BLOBs and CLOBs, and BFILES.

The JDBC API supports both two-tier and three-tier models for database access. In other words, JDBC can either be used directly from your application or as part of a middle-tier server application.

Two-Tier Model

In the two-tier model, a Java application interacts directly with the database. Functionality is divided into these two layers:

- **Application layer**, including the JDBC driver, business logic, and user interface
- **Database layer**, including the RDBMS

The interface to the database is handled by a JDBC driver appropriate to the particular database management system being accessed. The JDBC driver passes SQL statements to the database and returns the results of those statements to the application.

Three-tier Model

In the three-tier model commands are sent to an application server, forming the middle tier. The application server then sends SQL statements to the database. The database processes the SQL statements and sends the results back to the application server, which then sends them to the client. These are some advantages of three-tier architecture:

- Performance can be improved by separating the application server and database server.
- Business logic is clearly separated from the database.
- Client applications can use servlets access services.

The three-tier model is common in Web applications, where the client tier is frequently implemented in a browser on a client machine, the middle tier is implemented in a Web server with a servlet engine, and the database management system runs on a dedicated database server.

The main components of a three-tier architecture are as follows:

- **Client tier**, typically a thin presentation layer that may be implemented using a Web browser
- **Middle tier**, which handles the business logic or application logic. This may be implemented using a servlet engine such as Tomcat or an application server such as JBOSS. The JDBC driver also resides in this layer.
- **Data-source layer**, including the RDBMS

3.2 Design of Graphical User Interface (GUI)

E-Yearbook's main frame window is an AddressFrame class that extends a Java foundation Classes/Swing (JFC/Swing) JFrame. The AddressFrame is a container for other graphical components and also acts as a controller by handling various events generated by the child components. The child components are JPanel subclasses, each with a different responsibility:

- **AddressPanel** represents an address record. It also provides the UI for editing existing records and creating new records. It contains text fields for all the major properties of an Address object.

- **AddressActionPanel** provides buttons for all the major use cases that the application supports. This panel generates events that AddressFrame must handle. For example, when the user clicks Save, this panel generates an event. AddressFrame listens to and handles all important events from this panel.

- AddressListPanel provides a scrollable list of names on the far left of the AddressFrame. The list holds ListEntry objects. A ListEntry stores a database record's unique identifier. The record identifier (ID) allows the application to retrieve an entire record's contents into the AddressPanel.

The application uses a Data Access Object (DAO) to isolate the database-specific code. The DAO encapsulates database connections and statements. A DAO is a useful design pattern that allows loose coupling between an application and the underlying persistence-storage mechanism. The application's AddressDao class is an example of a DAO. When the AddressFrame edits, saves, or deletes an Address object, it always uses an instance of the AddressDao class. Although the Address Book application uses Oracle 10g database you could change it to use an entirely different database just by modifying this one class.

3.3 Integrating Oracle 10g with NetBeans IDE 5.0

Most IDEs provide a way to add libraries to the development classpath. Follow these instructions to add the Oracle 10g libraries to NetBeans IDE 5.5:

a. From the Tools menu, select Library Manager, as shown in Figure 2.

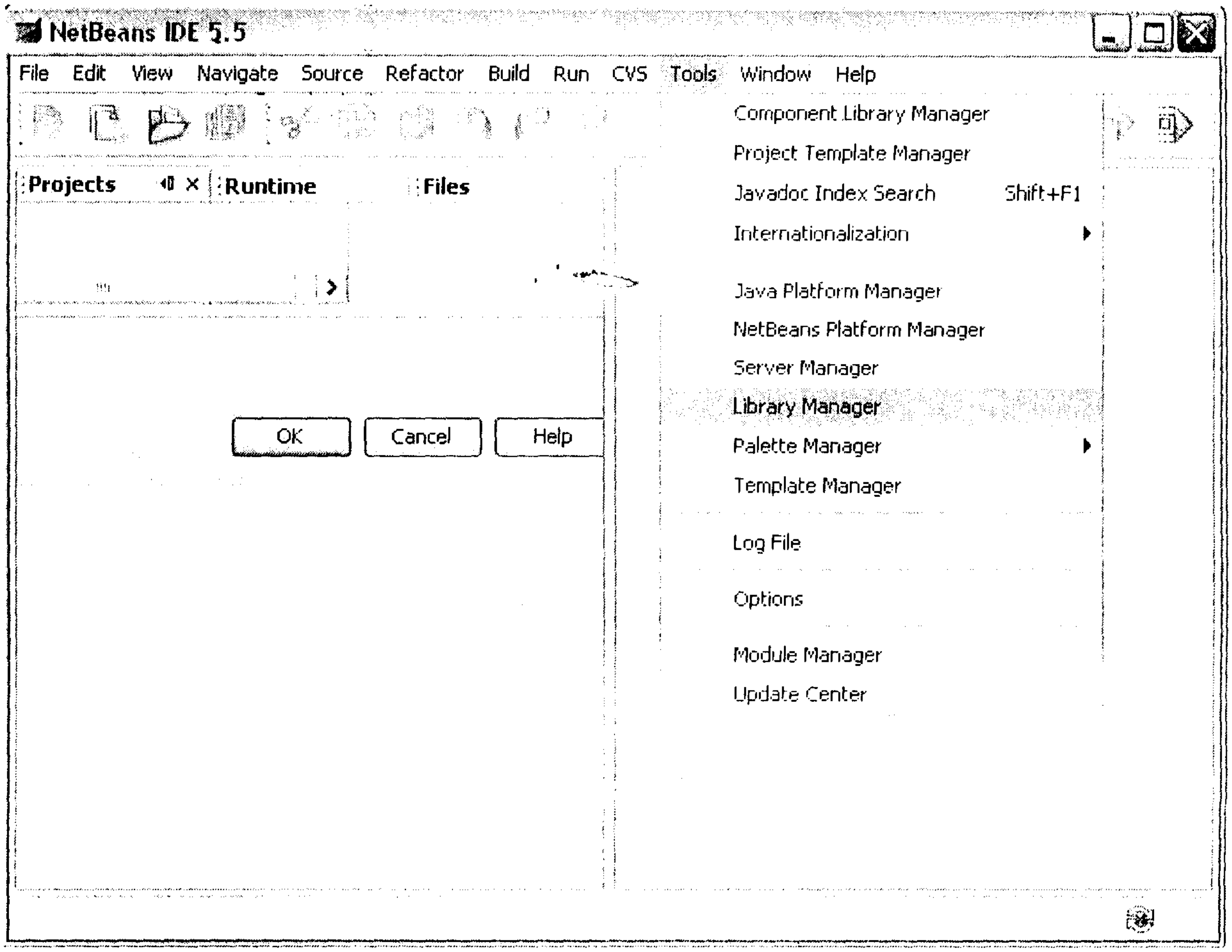
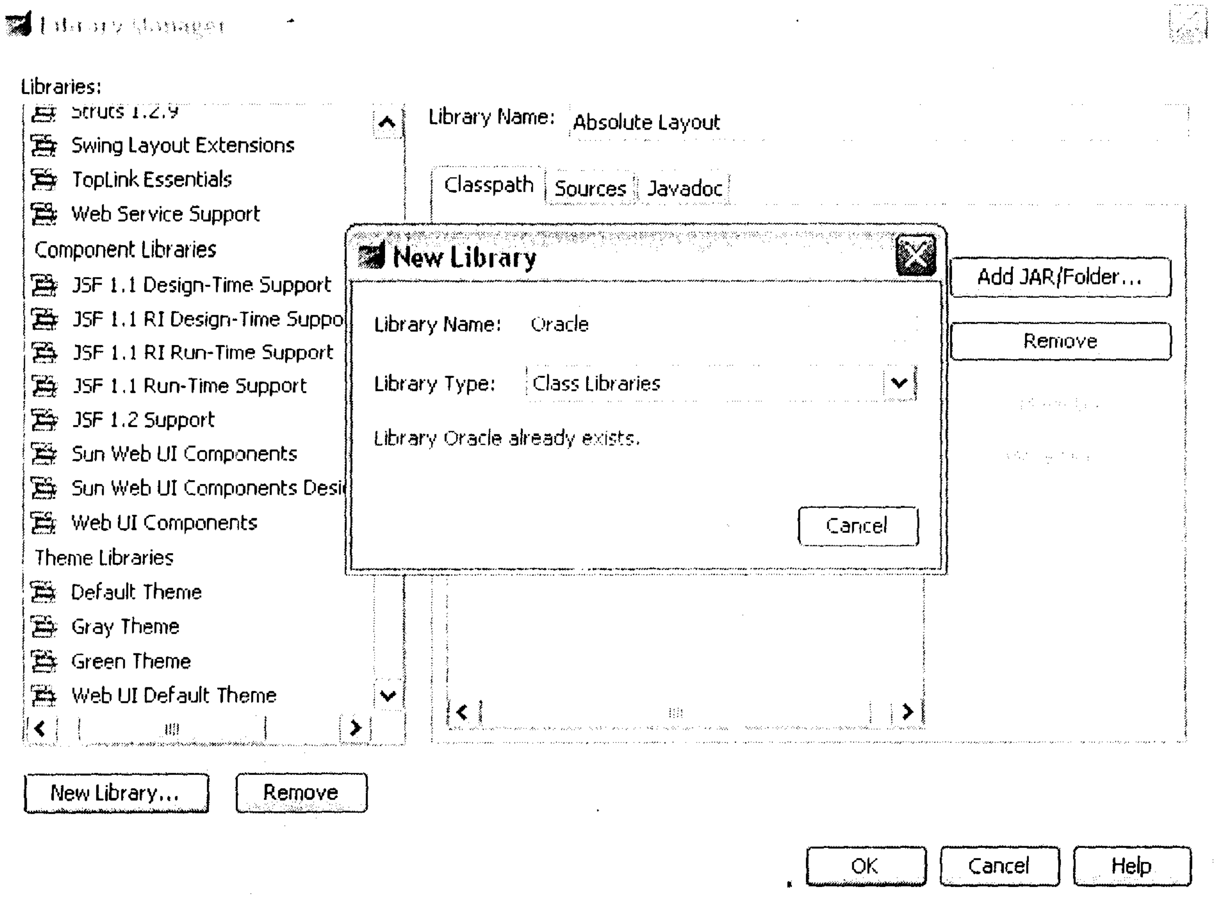


Figure 2: The library manager lets you add third-party libraries to your project.

b. In the Library Manager window, create a new library named Oracle, as shown in Figure 3. Click OK.



Library Manager

Libraries:

Library Name: Absolute Layout

New Library

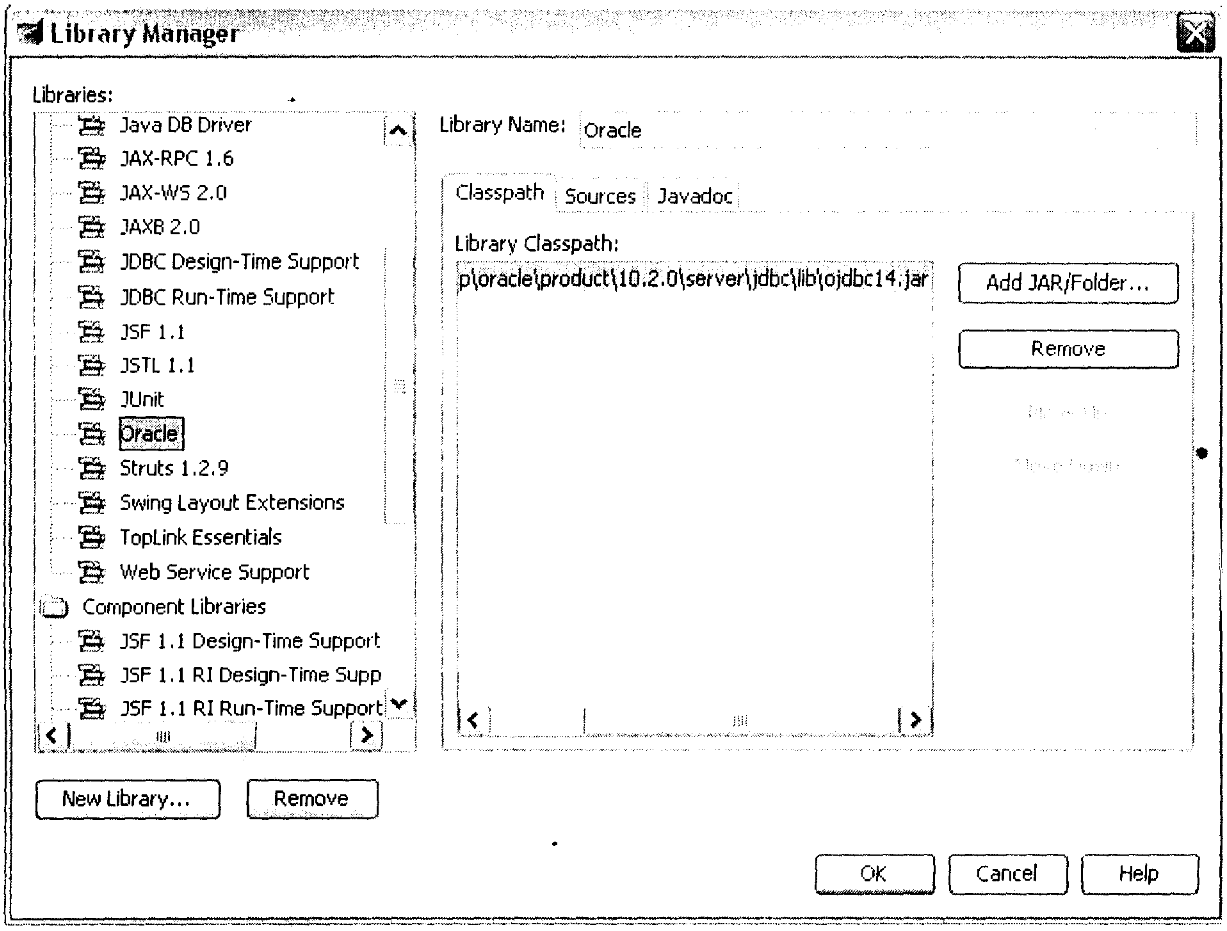
Library Name: Oracle

Library Type: Class Libraries

Library Oracle already exists.

Figure 3: Name the set of libraries that your project will need.

c. To add the ojdbc14.jar file to the Oracle library, click on Add JAR/Folder... in the Library Manager window. Navigate the file chooser to the ojdbc14.jar file in your Oracle 10g XE installation (C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib) and select it as shown in Figure 4



Library Name: Oracle

Library Classpath:

p\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar

You can now add the Oracle library to your NetBeans IDE 5.5 project by using the project's property settings. When you compile, debug, and run the application within the IDE, the IDE will be able to find the needed ojdbc14.jar file.

The E-Yearbook application reads the driver name from a configuration property file and passes the name to a loadDriver method. Additionally, as mentioned earlier, E-Yearbook encapsulates all database functionality into a Data Access Object (DAO), a core Java EE design pattern used to access data from a variety of sources. The DAO pattern works equally well for Java SE applications like E-Yearbook. The following code snippet shows how AddressDao reads the driver name and loads the driver:

```
private Properties bProperties = null;

public AddressDao(String addressBookName) {
    this.dbName = addressBookName;
    setDBSystemDir();
    dbProperties = loadDBProperties();
    String driverName = dbProperties.getProperty("oracle.driver");
    loadDatabaseDriver(driverName);
    ...
}

private Properties loadDBProperties() {
    InputStream dbPropInputStream = null;
    dbPropInputStream =
        AddressDao.class.getResourceAsStream("Configuration.properties");
    dbProperties = new Properties();
    try {
        dbProperties.load(dbPropInputStream);
    } catch (IOException ex) {
        ex.printStackTrace();
    }
    return dbProperties;
}

private void loadDatabaseDriver(String driverName) {
    // Load the Java DB driver.
    try {
        Class.forName(driverName);
    } catch (ClassNotFoundException ex) {
        ex.printStackTrace();
    }
}
```

3.4 Connecting to the Oracle 10g Database

A JDBC technology connection identifies a specific database and allows you to perform administrative tasks. Tasks include starting, stopping, copying, and even deleting a database. The driver manager provides all database connections. Retrieve a connection from the driver manager by providing a URL string that identifies the database and a set of properties that influence the connection's interaction with the database. A very common use of properties is to associate a user name and password with a connection. The Oracle 10g XE connection URL in the configuration properties file is:

oracle.url=jdbc:oracle:thin:@localhost:1521/XE

The ADDRESS table is created thus

```
"create table ADDRESS (" +
            "    REGNO          VARCHAR2(12) CONSTRAINT address_pk PRIMARY KEY," +
            "    SURNAME    VARCHAR2(30), " +
            "    FIRSTNAME   VARCHAR2(30), " +
            "    MIDDLENAME  VARCHAR2(30), " +
            "    PHONE1       VARCHAR2(20), " +
            "    PHONE2       VARCHAR2(20), " +
            "    EMAIL       VARCHAR2(30), " +
            "    ADDRESS1    VARCHAR2(50), " +
            "    ADDRESS2    VARCHAR2(50), " +
            "    CITY        VARCHAR2(30), " +
            "    SSTATE       VARCHAR2(30), ",+
            "    COUNTRY     VARCHAR2(30) " +
            ")";
```

The REGNO field is the primary key for each address record.

All remaining address record fields contain varchar elements of various lengths. For example, the LASTNAME field can contain a maximum of 30 varchar characters. The varchar type is equivalent to a UTF-16 Java char code unit. The Java technology code that uses the above SQL statement to create the ADDRESS table looks like the following code. The dbConnection is the same as the one shown in the previous code. We simply pass it into createTables, create a new Statement, and call the execute method to run the SQL code on the newly formed database. The strCreateAddressTable instance variable holds the SQL statement text.

```
private boolean createTables(Connection dbConnection) {
    boolean bCreatedTables = false;
    Statement statement = null;
    try {
        statement = dbConnection.createStatement();
        statement.execute(strCreateAddressTable);
        bCreatedTables = true;
    } catch (SQLException ex) {
        ex.printStackTrace();
    }

    return bCreatedTables;
}
```

CHAPTER FOUR

SOFTWARE DESIGN AND IMPLEMENTATION

4.1 Using the Database

Once the database and its tables have been created, your application can create new connections and statements to add, edit, delete, or retrieve records. In Address Book, these actions are controlled by buttons within the `AddressActionPanel`. Figure 5 shows the available options:

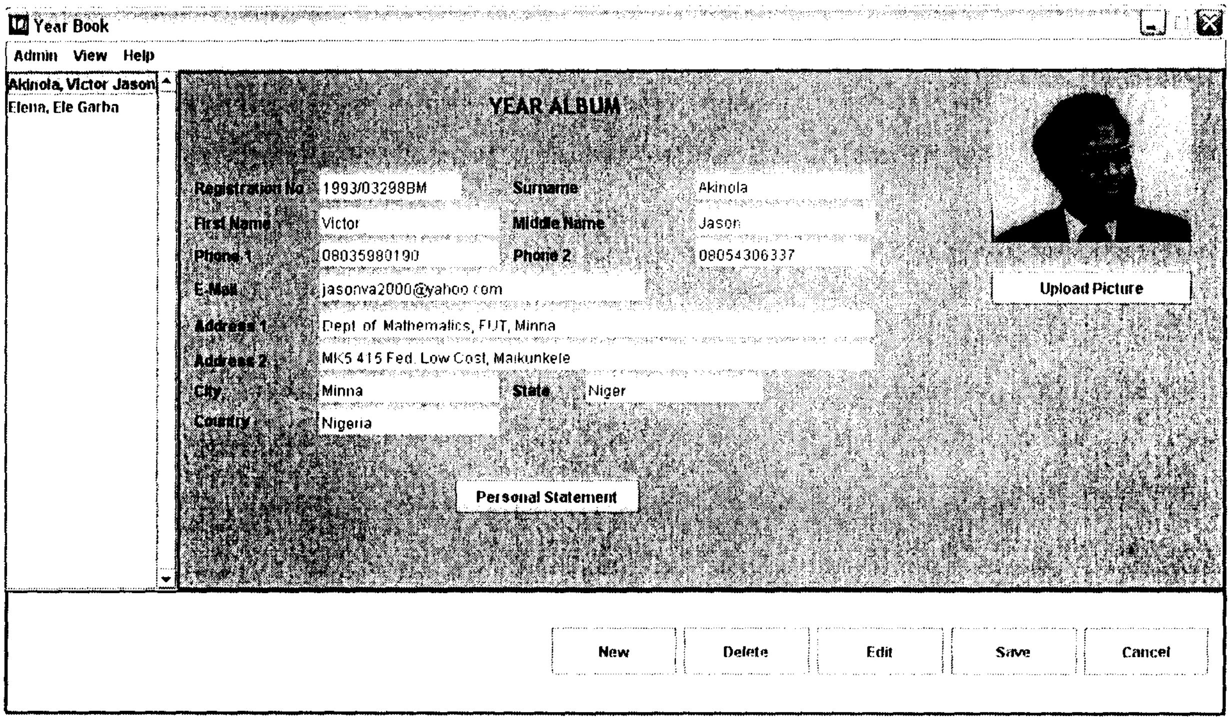
New. Create a new address record.

Delete. Delete the displayed address record.

Edit. Edit the displayed address record.

Save. Save the new or edited address record that is displayed.

Cancel. Cancel any edits or any attempt to create a new record.



Akinola, Victor Jason
Elena, Ele Garba

Registration No	1993/03298BM	Surname	Akinola
First Name	Victor	Middle Name	Jason
Phone 1	08035980190	Phone 2	08054306337
E-Mail	jasonva2000@yahoo.com		
Address 1	Dept of Mathematics, FUT, Minna		
Address 2	MK5 415 Fed. Low Cost, Maikunkele		
City	Minna	State	Niger
Country	Nigeria		

Figure 5: Address Book has several options for interacting with records.

The main window of the application is AddressFrame, which acts as a controller and as a view at the same time. It registers itself with the AddressActionPanel to receive notification when a user clicks anywhere on the action bar.

The New command clears the address entry panel and enables the user to edit all fields. No SQL commands are issued at this point, but the UI should allow you to enter a new address.

The Delete command attempts to delete the currently selected address record. AddressFrame retrieves the currently selected Address identifier from the AddressPanel and uses AddressDao to delete the record. The panel calls its own deleteAddress method, which calls the DAO's deleteRecord method with the correct REGNO. After deleting the record from the database, the application must delete the ListEntry from the AddressListPanel too.

```
private void deleteAddress() {
    int id = addressPanel.getId();
    if (id != -1) {
        db.deleteRecord(id);
        int selectedIndex = addressListPanel.deleteSelectedEntry();
        ...
    }
    ...
}
```

In the AddressDao, the deleteRecord method handles the actual deletion of the record from the database. The AddressDao creates a PreparedStatement when it first connects to the database.

```
stmtDeleteAddress = dbConnection.prepareStatement(
        "DELETE FROM APP.ADDRESS " +
        "WHERE ID = ?");
```

The PreparedStatement can be used multiple times, and this one uses a parameter to determine which record to delete. The deleteRecord method executes the update after setting the REGNO parameter:

```
public boolean deleteRecord(int id) {
    boolean bDeleted = false;
    try {
        stmtDeleteAddress.clearParameters();
        stmtDeleteAddress.setInt(1, id);
        stmtDeleteAddress.executeUpdate();
        bDeleted = true;
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    return bDeleted;
}
```

The Edit command allows the user to edit the currently selected Address record in the AddressPanel. For example, you can change the name, city, or phone number of a saved record.

The Save command retrieves either the newly created or edited Address from the AddressPanel and attempts to either update the existing record or create a new record. If the user has been editing a record, Save will update that record with the new information. If the user has created a new record, Save will insert a new record in the database. New records have not yet been saved. At this point, their REGNO field is still set to the default null value. This value changes to an autogenerated, unique record identifier once you save the record.

The following code in AddressFrame will save edited and newly created address records by calling the DAO's editRecord or saveRecord method, respectively. Of course, when you create a new record, the application must also update the AddressListPanel.

```
private void saveAddress() {
    if (addressPanel.isEditable()) {
        Address address = addressPanel.getAddress();
        int id = address.getId();
        if (id == -1) {
            id = db.saveRecord(address);
            address.setId(id);
            String lname = address.getLastName();
            String fname = address.getFirstName();
            String mname = address.getMiddleName();

            ListEntry entry = new ListEntry(lname, fname, mname, id);
            addressListPanel.addListEntry(entry);
         } else {
            db.editRecord(address);
        }
        addressPanel.setEditable(false);
    }
}
```

4.2 Deploying Your Application

Now that we have written the application, you must deploy it to users. Java technology applications can use a variety of deployment strategies, including Java Web Start software, applets, and stand-alone JAR files. We distribute the E-Yearbook application as a stand-alone application with JAR files.

The ANT build file, build.xml, uses a dist target to create MyAlbum.jar. It also places the database JAR file in the lib subdirectory directly under the MyAlbum.jar location. The final distribution structure for the application looks like this:

```
MyAlbum.jar
lib/ojdbc14.jar
```

If our build process includes classpath information in the ***MyAlbum.jar*** manifest file, you can run the application by simply passing the ***MyAlbum.jar*** file on the execution command line. On most platforms, you can also just double-click on the JAR file name in a graphical window. On a command line, you can use this simple execution command:

```
java -jar AddressBook.jar
```

This simple deployment and execution scenario can be accomplished by creating a manifest.mf file that becomes part of the MyAlbum.jar file. You can include information in the manifest that tells the Java programming language interpreter which class contains the main method and what other JAR files should become part of the classpath. The following manifest does both, and we can include it when building MyAlbum.jar.

```
Manifest-Version: 1.0
Main-Class: com.avj.AddressFrame
Class-Path: lib/ojdbc14.jar
```

Once our build process generates the application distribution structure shown previously, we can simply distribute this structure as a ZIP file. Users can simply unzip the file into any location and run the MyAlbum.jar file. The MyAlbum.jar file will contain the manifest file mentioned earlier and will tell your runtime environment what JAR files should also be on the classpath. Of course, Oracle 10g XE must be up and running in order for the application to work.

CHAPTER FIVE

SUMMARY/RECOMMENDATION

5.1 Summary

We have succeeded in developing a Java Application, with an oracle XE Database backend, that manages students' personal information for the purpose of sharing this with other students after graduation. Working with Oracle 10g XE is easy and fun. Oracle Database Express Edition is a relational database that stores and retrieves collections of related information. A database, also called a database server, is the key to solving the problems of information management. In a relational database, collections of related information are organized into structures called tables. Each table contains rows (records) that are composed of columns (fields). The tables are stored in the database in structures called schemas, which are logical structures of data where database users store their tables. In developing aqpplications that use the Oracle 10g XE database place the ojdbc14.jar file in your development environment's classpath so that your Java technology compiler and runtime environment can find the libraries to compile and run the application. Create a build process that places the ojdbc.jar file in a lib subdirectory immediately below your application's own directory and lastly add ojdbc14.jar to the application classpath by including a Class-Path property in your application JAR's manifest file.

5.2 Recommendation

The work can still be modified to become a three-tier application, that is, a web application. This will give some added functionality such as online checking of graduates, email services and so on.

REFERENCES

Oracle (2006) Oracle Database Express Edition 2 Day Developer Guide, 10*g* Release 2 10.2) B25108-01

John O'Donahue (2002) **Java Database Programming Bible** ISBN:0764549243 John Wiley & Sons © 2002 (702 pages)

Kevin Loney (2004) Oracle Database 10g: The Complete Reference. McGraw-Hill/Osborne

H.M. Deitel and P. J. Deitel (2005) Java How To Program 6th Edition. Prentice-Hall of India

Jason Price (2002) Oracle 9i: JDBC Programming. Oreilly

Jason Price (2004) Oracle database 10g SQL. McGraw-Hill/Osborne

R. Greenwald, R. Stackowiak, J. Stern (2004). Oracle Essentials: Oracle database 10g, 3rd Edition. O'Reilly