NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN OBJECT-ORIENTED VIEW OF BACKEND DATABASES IN A MOBILE ENVIRONMENT FOR NAVY AND MARINE CORPS APPLICATIONS

by

Lemuel Seth Lawrence
Kasey C. Miller

September 2006

Thesis Advisor:	Thomas Otani
Co-Advisor:	Arijit Das

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE An Object-Oriented View of Backend Databases in a Mobile Environment for Navy and Marine Corps Applications	**5. FUNDING NUMBERS** N/A
6. AUTHOR(S) Lemuel S. Lawrence and Kasey C. Miller	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

A Database Management System (DBMS) is system software for managing a large amount of data in secondary memory. The standard DBMS used today in both industry and the military is the Relational DBMS (RDBMS). The RDBMS is based upon the relational paradigm, whereas modern software development technologies that interact with the RDBMS are based upon the object-oriented paradigm. This difference in paradigms presents a conceptual mismatch which greatly reduces programmer and developer productivity.

Additionally, wireless handheld devices have become ubiquitous both in the military and in the community at large. These handheld devices provide a convenient means of information access. To date, the military has failed to capitalize on the use of handheld devices as a convenient means of information access with respect to the large amounts of information stored in its databases.

This thesis investigates various database application architectures and proposes an architecture that will not only overcome the conceptual mismatch between the relational and object-oriented paradigms, but also allows handheld device access to the database. A proof-of-concept prototype database application that provides handheld device access to a military personnel database is built to show the viability of the proposed architecture.

14. SUBJECT TERMS Object-Relational Mismatch, Relational DBMS, 3-Tier Database Architecture, Mobile Devices	**15. NUMBER OF PAGES** 87
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

AN OBJECT-ORIENTED VIEW OF BACKEND DATABASES IN A MOBILE ENVIRONMENT FOR NAVY AND MARINE CORPS APPLICATIONS

Lemuel S. Lawrence
Lieutenant, United States Navy
B.S. (Human Resource Management), New School University, 2000

Kasey C. Miller
Captain, United States Marine Corps
B.S. (Electronic Systems Technologies), Southern Illinois University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Authors: Lemuel Seth Lawrence

Kasey C. Miller

Approved by: Thomas W. Otani
Thesis Advisor

Das Arijit
Co-Adviser

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A Database Management System (DBMS) is system software for managing a large amount of data in secondary memory. The standard DBMS used today in both industry and the military is the Relational DBMS (RDBMS). The RDBMS is based upon the relational paradigm, whereas modern software development technologies that interact with the RDBMS are based upon the object-oriented paradigm. This difference in paradigms presents a conceptual mismatch which greatly reduces programmer and developer productivity.

Additionally, wireless handheld devices have become ubiquitous both in the military and in the community at large. These handheld devices provide a convenient means of information access. To date, the military has failed to capitalize on the use of handheld devices as a convenient means of information access with respect to the large amounts of information stored in its databases.

This thesis investigates various database application architectures and proposes an architecture that will not only overcome the conceptual mismatch between the relational and object-oriented paradigms, but also allows handheld device access to the database. A proof-of-concept prototype database application that provides handheld device access to a military personnel database is built to show the viability of the proposed architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A Database Management System (DBMS) is system software for managing a large amount of data stored in secondary memory, such as a hard drive. Among the different types of DBMSs, the de facto standard adopted and used by the vast majority of corporate organizations worldwide today is the Relational DBMS (RDBMS). RDBMS is built upon a solid theoretical foundation and allows the users to view and organize the data in an intuitive and easy-to-use tabular format. Since its introduction in the 1970s, the simplicity and theoretical elegance of the RDBMS accelerated its acceptance by database application developers.

During the last three decades, a number of proposals were made to either improve or replace the RDBMS, but none of them had gained any wide acceptance. Notwithstanding the improvements, the modern RDBMS in use today and the original RDBMS share the same core technology. This, in itself, is not a problem. However, the peripheral technologies, such as programming languages for developing database applications, and the computing environments, such as the wireless network and the Internet, have changed dramatically since the early days of the RDMBS. This causes a serious compatibility problem, and as a consequence, reduces the programmer productivity enormously. For example, developing web-based database applications such as the one for online shopping sites is very tedious, labor-intensive, and error-prone because the conceptual model, or paradigm, for web programs and the one for RDBMSs are very different. One of the primary objectives of this thesis is to study ways to fill the conceptual gap between software development technologies and RDBMS technology.

To narrow this study to a manageable size and complexity, the focus is limited to the database applications for the military. Different alternatives are investigated and a database application architecture that reduces the (negative) effect of the conceptual gap is proposed. As a part of this study, a proof-of-concept database application is developed that implements the proposed architecture.

The remainder of the chapter provides brief introduction to the following key technologies that are relevant to the proposed database application architecture: (1)

Object Oriented Programming Language, (2) DBMS in the Military, and (3) Wireless Environment for Database Applications. The chapter then concludes with a description of the organization of the remaining chapters of the thesis.

A. OBJECT-ORIENTED PROGRAMMING LANGUAGE

The primary software development technology in use today is the Object-Oriented Programming Language (OOPL). OOPLs are the industry leader for developing database applications because they provide developers with a more modern approach to development. Traditionally, software was developed using procedural languages based on the procedural paradigm that was linear and sequential. An OOPL is founded on the object-oriented paradigm that represents the world in terms of objects. This section describes the basic concepts of the OOPL and lists its advantages and disadvantages relative to database applications.

An OOPL is centered on creating and manipulating objects and defining the interaction between them. An object in an OOPL is an abstraction of a real world item, tangible or intangible, and is comprised of the data and functions that manipulate that data. The ability to define different types of objects and how those objects are manipulated allow the developer to very closely model the real world.

The advantages of utilizing an OOPL are object reuse, encapsulation, and inheritance. Object reuse is the ability to use the same object definition (or class) to represent multiple instances of the object. This reuse decreases not only the number of lines of code required to represent multiple objects, but also decreases the time necessary for developers to quickly produce applications. Encapsulation is described as hiding the internal workings of an object from the user of that object. For example, all OOPLs have some method of outputting text to the screen (e.g. System.out.println("text") in Java), however; the internal details of how that is accomplished are hidden from the database application developer. Encapsulation enables easier modification of program code and makes OOPLs simpler to learn compared to other programming languages. Additionally, inheritance is the ability to derive a new class from an existing one by extending or overwriting appropriate portions of the existing class. This, like reuse, decreases the number of lines of code required to represent several related objects. For example, if there is a class called Automobile then that class could be extended or modified to define the related classes of Car and Truck. The Car and Truck classes inherit properties and behaviors of the Automobile class such as Vehicle Identification Number, yet, extend the Automobile class by adding additional properties or behaviors as needed. These advantages of OOPLs provide greater flexibility and maintainability in database application software development.

The primary disadvantage to using OOPLs in database application software is a compatibility mismatch with the DBMS. A RDBMS is the standard approach in storing data, but OOPL objects do not directly map to the RDBMS's relational structure due to the RDBMS being limited to storing primitive data types (or in some instances user defined data types). An object, as defined earlier, is more than primitive data types; an object has an associated behavior or functions that do not directly map to the RDBMS. Thus, the inability to persist OOPL objects in their entirety is a compatibility mismatch between the RDMBS and the OOPL. This thesis will investigate ways to leverage the strength of OOPLs without incurring the burden of that compatibility mismatch.

B. DBMS IN THE MILITARY

DBMSs are heavily used in all functional areas of the military to store various types of information ranging from personnel and medical records to combat and intelligence data. The Department of Defense (DOD) has been storing data using DBMSs for years, specifically the RDBMS. RDBMSs have been used in the military because of their longstanding success in industry and are highly engrained into current military technologies. Though the military is highly reliant upon the RDBMS it presents limitations to both the military users and the database application developers.

From the user's perspective on data access to the database, the DBMS is hidden or transparent. The user interacts with the database application to store and retrieve data with no understanding of how that data is stored, retrieved, or managed. Current military database applications have targeted the Personal Computer (PC) as the primary means of user interface. However, current trends in technology have progressed to small, convenient technologies known as mobile devices, i.e., the Personal Digital Assistant (PDA) and smart phones. Likewise, the military has recently begun procuring a growing number of mobile devices for both tactical and non-tactical purposes. Though these mobile devices are becoming readily available for military use, they have not been utilized to access military database applications. For example, key military personnel often carry PDAs for scheduling, email access, and storage of important information. If that information is not resident on the PDA it is most likely in a database application. Assuming the PDA is capable of storing all of the information requested from the database application, how is that information displayed to the user? In the case of a military RDBMS, the primary means of display is in the form of table(s). This in itself is a potential limitation to the user because of the device's limited display capability and potential for large tables returned from a military RDBMS.

Additionally, the heavy military use of RDBMSs present limitations to the database application developers. The database application developer must develop the database application around the heavily used RDBMS. As previously stated, the primary software development technology in use today is the OOPL. This combination of heavily used RDBMSs in the military and modern software development technologies utilizing OOPLs causes the developer to encounter the compatibilities mismatch previously discussed. Military database application developers have three means of circumventing this mismatch. First, the developers may undertake the tedious process of manually mapping the data object's contents to the RDBMS. Second, the developers may use older software development technologies not dependent on OOPLs. However, these older software development technologies are not always capable of cleanly modeling real world problems when compared to OOPL software development technologies. Third, developers may utilize a DBMS that is capable of storing objects. This compatibilities mismatch limitation has become more and more apparent as software development technologies continue to become more OOPL based and military database applications continue to relay on the RDBMS.

The need to update DBMS technology in the military is imperative in order to extend the use of software development technologies like OOPLs to database applications both currently used in the military and those that are being developed. An increased use of software development technologies in military database applications will shorten the applications development time and improve maintainability and reliability. This thesis seeks to make use of software development technologies and mobile devices in order to improve remote data access in a military database application environment.

C. WIRELESS ENVIRONMENT FOR DATABASE APPLICATIONS

People use wireless devices (to include handheld devices) more and more. These are ubiquitous devices, so it would be ideal if database information could be provided to those devices. Further, wireless devices are becoming OOPL enabled. These OOPL enabled wireless devices provide the database application developer a means of incorporating wireless devices into database applications using modern development technologies. Furthermore, the combination of OOPL capable wireless devices and the growing demand for wireless access to data justifies the incorporation of wireless devices in database applications. By providing database application data to wireless devices the value of the data increases. The demand for remote data access and the emerging mobile wireless devices that make that access possible are a direct correlation to the data's increase in value. Joining wireless technologies with database applications has its advantages and disadvantages.

The primary advantage of using a wireless device, or handheld, in database applications is information availability for the user. This increased information availability provides access to large stores of data where and when it is needed. For example, a growing number of military police forces have incorporated handheld wireless devices. This provides remote access via the handheld wireless device to vast amounts of information on any individual that military police forces may encounter. Another advantage of incorporating wireless devices into database applications is increased scalability. Adding wireless devices to an existing network requires little to no additional infrastructure, yet greatly enhances the number of potential users. The increase in both scalability and availability provide a means to improve user productivity.

Though wireless devices provide advantages, there are some disadvantages. In addition to the limitations of mobile devices previously mentioned, there is a connection difference between a wired and wireless device. The wireless connection of a handheld device provides concerns in the way of reliability and security. Reliability is of concern because the wireless connection is more susceptible to interference than a wired connection. Similarly, data security is a concern for corporations of all sizes. The introduction of wireless access and transmission of that data only enhances this concern further due to removing the traditional network security perimeter. Though there are disadvantages to wireless devices in database applications, there is an undeniable demand for remote access to data. This thesis will further investigate the use of wireless devices in database applications.

D. ORGANIZATION

The organization of this thesis is as follows: Chapter II provides the background on database application designs including a discussion of types of DBMS, types of software architecture for database applications, types of software client applications, and types of client connectivity. Chapter III provides decisions on the database application software architecture, the type of backend DBMS, client connectivity, and on type of database view. The chapter then concludes with proposed database application software architecture. Chapter IV opens with a discussion of the application domain and provides an introduction to the proof of concept. The chapter then provides a description of the prototype application developed as a result of this thesis. Finally, the chapter concludes with findings during implementation. Chapter V provides an evaluation of the implemented solution with respect to the conceptual gap posed by this thesis. Further, this chapter provides a discussion of the general findings and possible extensions on the existing implementation and recommendations for follow-on research.

II. BACKGROUND

A. INTRODUCTION

In this chapter, background information will be provided in the following four areas: (1) Types of Database Management Systems, (2) Types of Software Architectures for Database (DB) Applications, (3) Types of Software Client Applications in a Three-Tier Architecture, and (4) Types of Client Connectivity. A discussion of each area will include a general introduction to the concept followed by a short discussion of the pros and cons relative to each individual area of concern.

B. TYPES OF DATABASE MANAGEMENT SYSTEMS

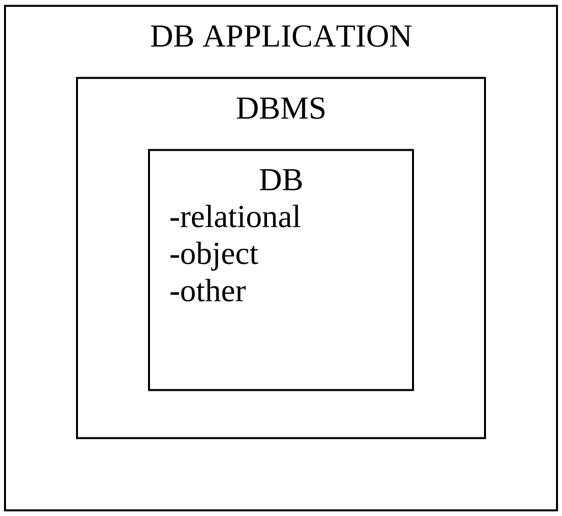


Figure 1. DB Application Architecure

A DB application as shown in Figure 1 is software that manages information stored in secondary memory devices. DB applications consist of three major components, the user interface, the application logic, and data storage. DB applications can be written completely by using programming languages such as C, C++, or Java. However, a typical DB application will utilize system software called a Database Management System (DBMS) to handle the tedious task of data operations, updates, retrievals, and deletions. Figure 1 above shows that the DB application hides the DBMS and the DB from the user regardless of the type of DBMS that the DB application implements. The DBMS is software that utilizes a DB (this DB can be relational, object-oriented, etc) to store data and provide a means of accessing that data in a standard format such as Structured Query Language (SQL). The DBMS is utilized by the DB application to store and access data respective to the specific DB application’s need, and in doing so provides some degree of transparency and independence as to how the data is actually stored. Currently, there are three prominent DBMSs; (1) the object-oriented DBMS, (2) the relational DBMS, and (3) the object-relational DBMS.

1. Object-Oriented Database Management System

Object-Oriented Database Management Systems (OODBMSs) provide persistent storage of objects. These objects can consist of primitive data types or other objects which may in turn consist of primitive data types or further objects and so on. Additionally, these objects may be defined via an object-oriented programming language such as Java, C++, or Smalltalk, but a true OODBMS provides persistent storage of objects independent of specific programming languages. OODBMSs are generally used in applications where the object-centric point of view is appropriate. Further, an OODBMS is best suited when user sessions consist of retrieving at most a few objects and then manipulating or processing some portion of those objects for an arbitrary period of time. Objects can be of arbitrary size and complexity so applications that do not require extensive queries yet have complex objects work well with OODBMSs.

There are limitations when implementing the OODBMS solution in DB applications. OODBMSs have not gained popular support in industry today. The lack of industry support has led to very few OODBMS developers and vendors to produce robust and applicable OODBMSs. Additionally, the traditional Relational DB Management System (RDBMS) is suited for nearly all industry requirements and OODBMSs have only caught a very small niche of the market to include computer-aided design (CAD) and telecommunications. The OODBMS’s small market share is most evident when comparing the 1999 sales revenue where object-oriented systems had only $211 million in sales where the combined sales of the relational and object-relational DBs had a staggering $11.1 billion in sales by comparison (Leavitt Communications, 2000). Further, OODBMS applications do not fit well in query extensive environments due to the potential for large and complex objects to be stored in the DBMS. A complex query where several large objects are returned is a costly operation (in terms of time and efficiency) and therefore OODBMSs are not an ideal solution in applications that require several queries or an environment that requires a single query to return several related objects (Ramakrishnan & Gehrke, 2003). To further complicate design and implementation, there is currently no established theoretical definition of the object-oriented data model (Bertino & Martino, 1993).

Overall, the OODBMS fits well in specific problem domains that do not require extensive queries. However, if a DB application must convert its existing data (most likely in a RDBMS) in order to make use of an OODBMS then an OODBMS is not a practical solution. As Bertino and Martino more eloquently stated,

> Realistically, a number of factors has to be taken into account: it is impossible to abandon, from one day to the next, the ‘old’ DBMS, due to the obvious effects on a company’s operating continuity, the shortage of suitably qualified staff, the lack of real ‘guarantees’ that it will be possible to reuse new data and applications environments already created, and ultimately to preserve existing investment intact.
>
> (Bertino & Martino, 1993)

These factors are reason enough to sway away from OODBMSs as an implementing solution in instances of trying to utilize existing databases or to upgrade an existing DB application to meet the object-oriented trend in modern technologies. The OODBMS is a viable option in one of the niche environments or in an object-oriented application that does not rely on any existing, legacy, DBMSs. Furthermore, the OODBMS is a prime solution for implementing a DBMS from the ground up.

2. Relational Database Management System

Originally proposed in 1970 by Edgar Codd the relational model is now present in all ranges of systems from Personal Computers (PCs) to large server applications and is clearly the dominant means of storing all types of data. The dominance of the relational model is in part due to the model’s mathematical foundation and relatively longstanding use, passing the test of time (Elmasri & Navathe, 2004). The relational model is well suited to store most types of data and works well if the relationships between data are not too complex. A relation in a Relational Database Management System (RDBMS) is stored in the form of tables consisting of rows and columns of primitive data types. These rows and columns represent the data itself or the relationship between tables that can be rather simple or very extensive allowing the RDBMS a wide range of scalability relative to the amount and types of data it contains. When compared to other database technologies the RDBMS is much more mature and is clearly the dominant persistence mechanism on the market today. Additionally, the RDBMS has several well established vendors, and existing standards such as SQL and Java Database Connector (JDBC) (Ambler, 2006). For these reasons the RDBMS is well suited for nearly any DB application or task in potentially any problem domain related to data storage and access.

Though the RDMBS will clearly work in nearly all problem domains, it has inherent drawbacks. First, the RDBMS will not explicitly allow for the storage of a user defined data type or object. Second, in the case of large databases, queries and searches become slow and computationally intensive. Lastly, the relational model does not map well to all real world applications such as CAD applications. The inability to map well to certain applications is due to the RDBMS being somewhat two-dimensional, as each table only has two dimensions, namely the row and column.

The RDBMS is clearly a viable option in any number of problem domains due to its extensive range of applicability and strong mathematical foundation. The RDBMS clearly has many advantages but exhibits drawbacks as well, such as strong coupling and an inability to map to all real world applications as already discussed. Further, the RDBMS has a lack of ability to relate directly with the modern object-oriented programming paradigm languages such as Java. These pros and cons must be weighed but an obvious and unavoidable fact is that RDBMSs are everywhere and in most cases unavoidable.

3. The Object-Relational Database Management System

The object-relational database concept has been heralded as the next great wave in DB technologies by Michael Stonebraker, Chief Technology Officer at Informix(Leavitt Communications, 2000). A simple description of the object-relational concept is a relational DB that is capable of storing not only primitive data types, but also objects. This next great wave is founded by Stonebraker’s claims that Object-Relational DBMSs

(ORDBMSs) have four essential properties: (1) support for base type extensions in an SQL context (provides for user defined data types that are capable of utilizing the already present primitive data types and a means of querying them via an SQL type standard), (2) support for complex objects in an SQL context (provides a means of querying objects that contain other objects or lists of objects that in turn are comprised of primitive data types or other objects, again via an SQL type standard), (3) support for inheritance in an SQL context (provides a means of querying objects that expand/extend another object's basic structure, to include functions and data, via an SQL type standard), and (4) support for a production rule system (a production rule here is a simple "on event - do action" rule utilized to maintain the integrity of the DB generally referred to as a trigger in traditional DBs) (StoneBraker & Moore, 1996). In theory these properties combine the longstanding support and existing standards that the RDBMS has gained over time with the desired object-oriented design principles (such as extensible data types, function and data inheritance, etc).

ORDBMSs add object oriented features to the relational concept and provide the ability to navigate objects in addition to a RDBMS's ability to join tables that now potentially contain objects. By implementing both objects and relations in a DBMS, an ORDBMS can execute complex analytical and data manipulation operations to provide user defined functions on the stored data. ORDBMSs also support the robust transaction and data-management features of a RDBMS while at the same time offer a limited form of the flexibility provided by the object-oriented design concept from the OODBMS. The relational foundation of the ORDBMS permits tabular structures, Data Definition Languages (DDLs), and data that is accessible via familiar approaches such as SQL, JDBC, and potentially user defined call interfaces via the object-oriented programming paradigm(Ambler, 2006).

The ORDBMS is not without its disadvantages. First, ORDBMSs do not provide a means of efficient access to its data. Even though ORDBMSs are capable of storing complex data types they are inefficient in their retrieval of such data due to that data complexity. The complex data types are often queried via an Object Query Language (OQL) that is not mathematical in its foundation and therefore is very difficult to optimize in comparison to SQL. Second, combining the relationships present in a

RDBMS and relationships present in an object or between objects seems logical. These two types of relationships (RDBMS and object) do not necessarily complement each other. For example, objects can be related via other objects; where as, the traditional relation in a RDBMS would require another table (or relation) to express this same relationship. The ORDBMS potentially represents the same relationship more than once; internal to the objects and in the tabular structure of the ORDBMS. Thus, the ORDBMS potentially has redundant representations of the same relationship. These two different methods of representing a relationship (RDBMS and object) lead to the concept of an Object Relational Mismatch (or Impedence Mismatch) which is further described in Chapter III.

C. TYPES OF SOFTWARE ARCHITECTURES FOR DB APPLICATIONS

DB application architectures consist of one of three possibilities, (1) Single-Tier, (2) Two-Tier, or (3) Three-Tier. The DB application has three distinct software segments or layers; specifically, these are the (1) Presentation, (2) Business or Application Logic, and (3) Data Access layers. The manner in which these three layers are composed differentiates the three DB architectures. Each of these architectures will be discussed and their advantages and disadvantages explored further.

1. Single-Tier

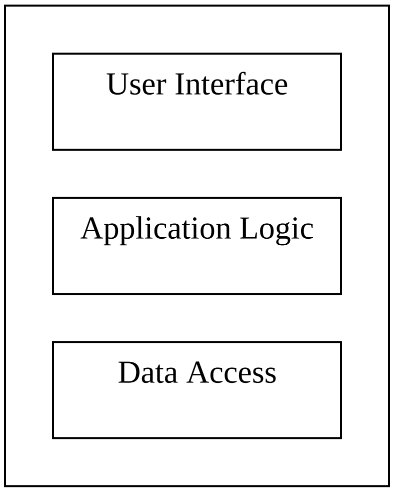


Figure 2. Single-Tier Architecture.

With the Single-Tier architecture, as illustrated in Figure 2, a single piece of software includes the user interface (presentation layer), the application logic layer (business logic), and the data access layer. Traditionally, this application would run on the mainframe and users would access it on a mainframe itself or through dumb terminals that could only perform data input and display functionality (Ramakrishnan & Gehrke, 2003). This traditional approach is the single-tier architecture because the presentation software, the application logic software, and the data access software are all resident on a single machine. A simplistic example of a modern single-tier DB application would be a Microsoft Access DBMS that resides on a single Personal Computer (PC) and is accessed only from that PC.

This simple architecture has several advantages. The single-tier architecture is easily and centrally maintained by a single or very few system administrators. It also captures all layers of complexity within a single point for ease of access to the data for both the end user and the developer. Further, this simple architecture allows for timely development of relatively serious and robust applications without incurring the enormous effort and long development cycles that are often the norm for mainframe development.

The most serious disadvantage of the single-tier architecture is the lack of scalability. Traditionally, the single-tier architecture is composed of a single DB and provides access to a single user. Thus, by definition the single-tier architecture does not scale to a large number of end users.

2. Two-Tier

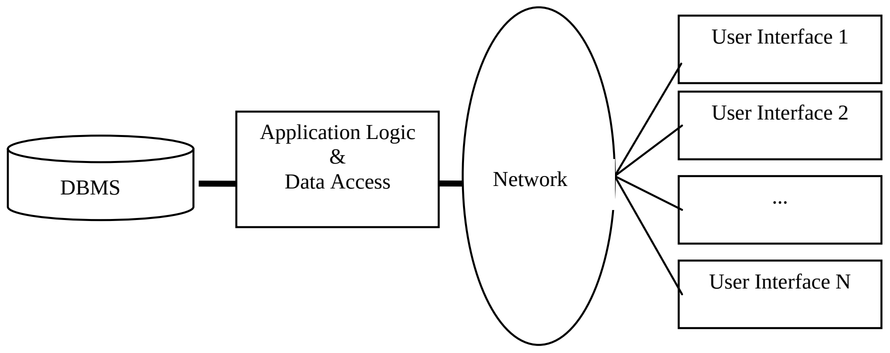


Figure 3. Two-Tier Architecture

Figure 3 above shows a two-tier architecture that separates the client (User Interface) from the database server (application logic and data access). The two-tier architecture is also commonly referred to as the client-server architecture. The means of separating the complexity between the server and the client can vary. Traditionally, the user interface is merely a GUI program whereas the application logic (often referred to as business logic) and data access layers are combined and maintained via an additional and separate program. This particular separation of complexity is known as a thin client. Conversely, the other means of separating complexity is where the business logic and GUI are contained in a single program and the data access layer of the architecture is contained in a separate program. This variation of the two-tier architecture is commonly referred to as a thick client.

The two-tier software architecture has a few important advantages. The two-tier architecture DB application is quickly developed using modern tools such as Microsoft Visual Basic or Sybase Powerbuilder. Moreover, it allows for separation of complexity by separating the presentation issues from the data management issues to allow for ease of maintaining the software and increase the scalability of the system to multiple end users utilizing either a thin or thick client. The two-tier architecture's largest advantage over the single-tier architecture is the increased scalability.

There are a few inherent drawbacks specific to the thick client compared to the thin client. The first drawback of the thick client is the lack of a central location to update and maintain the business logic of the DB application since the business logic is running on multiple clients. Second, the DB must rely on the client to leave the data in a safe or acceptable state following any transactions, thus providing greater possibility of error and more complexity. Lastly, thick clients do not scale well with a large number of clients due to potentially large data transfers (queries) causing a potential bottleneck. Further, the scalability is compounded when multiple databases are considered. For each client there can be N connections open with the server where N is the number of databases which clearly does not scale well with multiple clients and DBs in the DB application as a whole.

The drawbacks of the thick client two-tier architecture have led to the primary use of thin to extremely thin clients. Today, these thin clients can consist of merely a web browser that connects to a DB (ultra-thin client) or a user defined application such as a Java application (Ramakrishnan & Gehrke, 2003). Clearly, the two-tier architecture is more scalable to more problem domains than the single-tier architecture, however; the scalability is still a concern for widespread use depending on the number of users, the number of databases, and the amount of data transferred on average per transaction between the server and the client.

3. Three-Tier

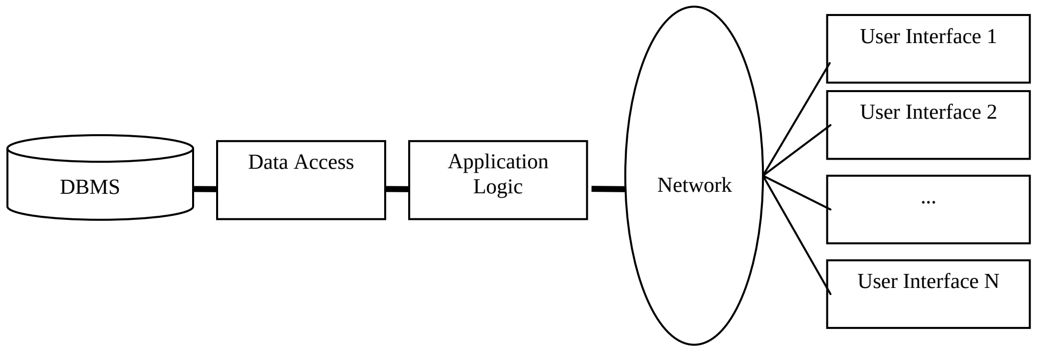


Figure 4. Three-Tier Architecture

Like the two-tier architecture, the three-tier architecture separates complexity. However, the three-tier architecture separates the application logic from the data management issues as shown in Figure 4 above. According to Ramakrishnan's and Gehrke's Database Management System, this architecture allows for three distinct tiers or layers: The Presentation Tier (User Interface), the Middle Tier (application logic layer), and finally the Data Management Tier (data access layer). The Presentation Tier provides the users of the DB application with an interface to make requests (query), provide input, and to see results of those inputs/requests. This specific tier is either a full software application or a web based application in most of today's DB applications. The Middle Tier executes the application logic and is generally programmed in a language such as Java or C++. Lastly, the Data Management Tier is the data access portion normally implemented by using a DBMS, but the use of a DBMS is not a requirement. These tiers have some means of communicating via a standard or customized protocol. Specifically, the means of communication between the presentation layer and the application layer is normally web based, and therefore, Hyper Text Transfer Protocol (HTTP) or another well defined protocol is used. Likewise, the application and data management layers communicate via a standard interface such as Java Database Connector (JDBC).

The three-tier architecture has five advantages: (1) Heterogenous System, (2) Thin Clients, (3) Integrated Data Access, (4) Scalability to Many Clients, and (5) Software Development Benefits. A Heterogeneous System allows the applications to utilize the strengths of different hardware platforms at their respective tiers. Secondly, Thin Clients allow for the presentation layer to be handled on as light a client as possible and not have to maintain the integrity of the data on the client side making this architecture much more scalable than the one-tier or the two-tier architecture. Third, Integrated Data Access allows for all the accesses to the data layer to be handled at the middle tier further separating complexity. Fourth, the entire three-tier architecture is extremely scalable to multiple clients. This scalability is enabled by both the thin client concept and the ability to place multiple systems (here systems refers to hardware systems) at any potential bottleneck. Lastly, the three-tier architecture has inherent software development benefits due to being logically split up into layers that correspond to presentation, business logic, and data management. Further, the three-tier architecture allows reusable software components at each layer and the use of well defined protocols or APIs allowing for a loose coupling between components.

There are a few drawbacks with the three-tier architecture. First, the three-tier architecture is more complex and therefore is more prone to errors and mistakes in development, however; most of this is mitigated by using defined APIs or protocols between the layers of the architecture. Second, the DB application must have some notion of state across the layers. Each layer must be aware of the state of each bordering layer in order to allow for efficient and correct access. Again, this is mitigated via APIs and well known protocols. Lastly, though the DB application is broken up and logically allows for the use of the respective software or hardware components it potentially requires much more than a single or small group of administrators to maintain it (Ramakrishnan & Gehrke, 2003).

D. TYPES OF CLIENT SOFTWARE APPLICATIONS IN A THREE-TIER ARCHITECTURE

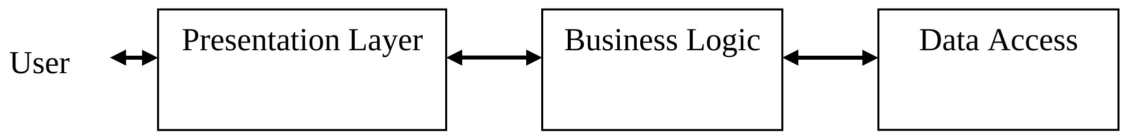


Figure 5. Basic Three-Tier Design

As Figure 5 above shows, the presentation layer is the layer that interfaces with the user and it is therefore vital that the correct implementation of this layer be utilized for both functionality and aesthetics. There are three distinct methods of presenting data at the presentation layer in a three-tier architecture DB application, (1) a pure web browser, (2) a Java applet, and (3) a full software application. Each of these approaches will be explored further and the advantages and disadvantages of each discussed.

1. Pure Web Browser

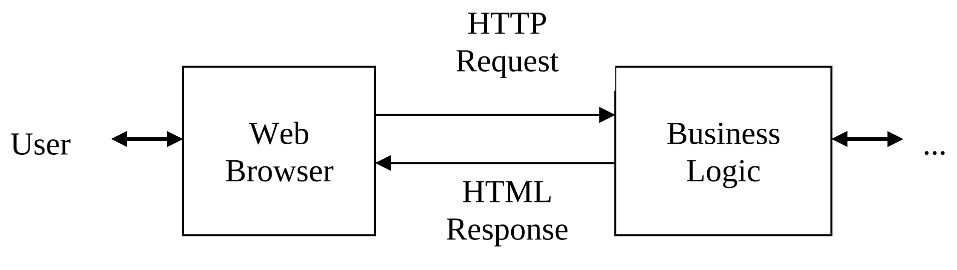


Figure 6. Pure Web Browser Presentation Layer

The true web browser client, as shown in Figure 6, provides a simple yet potentially robust means of displaying results while requiring little to no data processing by the client (all it does is display data to a screen). This provides for scalability and decreased maintenance in the client software. Further, this approach uses a well known protocol, Hyper Text Transfer Protocol (HTTP), to request the data providing for an overall accepted and well defined communications media. The additional benefit of utilizing the well defined protocol approach to displaying data, namely Hyper Text Markup Language (HTML), is that the only items being transferred between the presentation layer (the client) and the middle layer is HTML data that can be viewed in any compatible browser, i.e. Internet Explorer or Mozilla. The use of an existing browser provides the advantage of requiring no additional software on the part of the client. Further, the client has no need for insight into the middle tier or the data management tier to serve its presentation function thus achieving the separation of concerns in the application's three tier architecture.

The web browser approach does have one severe limitation. The web browser is limited on its ability to display complex data because it is restricted to use of simple HTML forms, javascripts, Java Server Pages (JSPs), etc. Though this is a somewhat robust means of displaying data it is still lacking the true display power of programming language. The web browser is insufficient if the client is required to display complex data.

2. Java Applet

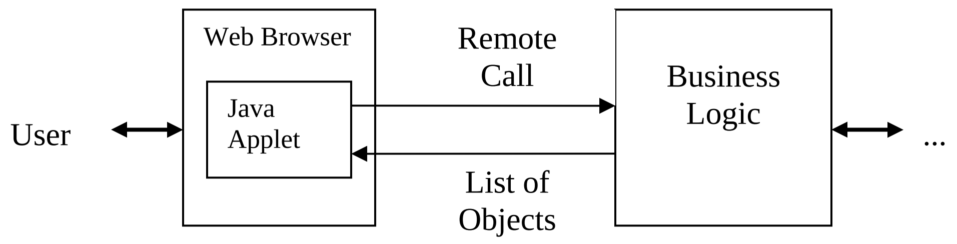


Figure 7. Java Applet Presentation Layer

The second method of presenting data to the user is the Java Applet as shown in Figure 7 above. The Java Applet ist defined by Microsoft online as:

> …a Java class that is loaded and is run by a Java program that is already running, such as a Web browser or an applet viewer. Java applets can be

downloaded and run by any Web browser that can interpret Java, such as Microsoft Internet Explorer, Netscape Navigator, and HotJava. Java applets are frequently used to add multimedia effects and interactivity to Web pages, such as background music, real-time video displays, animations, calculators, and interactive games. Applets can be activated automatically when a user views a page, or they may require some action on the part of the user, such as clicking an icon in the Web page.

(Microsoft TechNet, 2005)

The Java Applet approach to displaying data adds the robustness of the programming language to the web browser by embedding the program into the web browser itself. Further, it does this without adding the complexity to the software client. The Java Applet itself is maintained by the business logic layer and retrieved when the data is to be accessed via the browsers interface to the applet. Since the program is retrieved from the business logic layer vice the software client, the software maintenance required is reduced due to the applet only being permanently resident in one location.

The largest drawback to the Java Applet is that it has limited access to the client's hardware resources. This limitation is a necessary security feature that limits the ability of the Applet to store the data on the client hardware. Traditional software resident on the client hardware does not suffer from this limitation. Therefore, in the case that maximum data access and access to client system resources are required, a full application must be employed.

3. Full Software Application

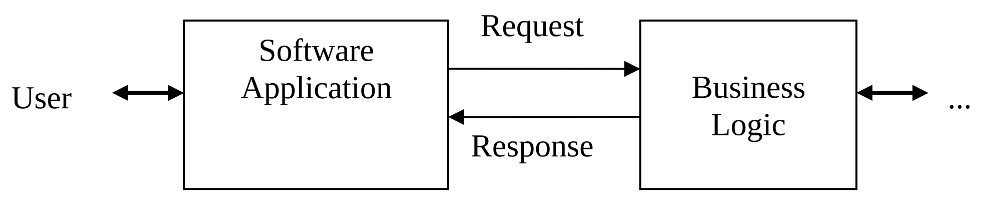


Figure 8. Full Software Application

To gain the full program complexity and the display capabilities that go with it, the full software application, as shown in Figure 8 above, is the primary means utilized today. The application is not limited by the security aspects as the Java Applet is and therefore, it will have full access to the client's hardware resources (or some user defined level of access). The user defined application concept has benefits, but there are a few drawbacks.

The first drawback is that software maintenance becomes much more difficult compared to the other approaches because that software resides on all hardware clients. This makes software maintenance or modifying the application difficult at best because all hardware clients that maintain a version of the software must be updated. Lastly the full software application approach is much more computationally dependent upon the client hardware since the program is being run directly on the client machine. This is a very attractive means of viewing backend databases due to having potentially full view into the data portion of the architecture. However, this could also serve as a hindrance due to potential maintenance problems in a large scale deployment type of environment where multiple clients are unavoidable.

E. TYPES OF CLIENT CONNECTIVITY

The way that the presentation layer (or software client) interacts with the rest of the architecture can be either hardwired or wireless. Each of these two means of connectivity between the presentation layer and the rest of the system has its benefits and drawbacks. Here, those benefits and drawbacks are briefly discussed in the context of DBMS architectures.

1. Wired

A wired client provides many benefits. First, the clients are all locally administered and maintained. Second, the wired client provides faster access to data than a wireless client. This becomes apparent in the case of large amounts of data being transmitted between the client and server where the wireless client would suffer by comparison. Third, the wired client provides for increased control over the security of data. This is largely in part due to the ability to monitor all access points that are defined by hardware (via a well defined perimeter) in the architecture whereas a wireless client could potentially access the data from anywhere. Lastly, the wired client permits the utilization of existing infrastructure. This makes possible efficient and potentially more cost effective solutions to connectivity.

The wired client also has several drawbacks. The largest and most apparent drawback is mobility. Wired clients are severely limited when it comes to mobile access to the application. This potentially restricts users' ability to access data in a timely fashion as they must be physically connected to a data point. Another disadvantage of wired clients is that they either require pre-existing infrastructure or they become rather costly to implement and build from scratch depending on the system infrastructure requirements (i.e. transmission media requirements such as fiber vs. twisted pair). These drawbacks lead to the requirement for either a truly wireless application or a combination of wired and wireless clients allowing for the benefits of the wired client to those users it is available for and the mobility to the clients that require it.

2. Wireless

The wireless client is a necessity in an environment that requires mobility; however, there are benefits to the wireless client beyond mobility. Wireless clients allow for a quick and easy way of building the system from scratch; all that is required is a server with a Wireless Access Point (WAP). The use of a WAP permits quick and potentially cost effective deployment of a DB application. Further, the wireless client offers a more convenient means of accessing information stored in a DB application. Yet another benefit is the simple fact that wireless access to data is in demand in nearly all types of systems and applications. Thus, the wireless client meets that demand by providing the coveted mobility and usability to the user but it is not without costs.

The costs of wireless access are not necessarily monetary; they are in the realm of the security of the data. The wireless environment, unlike the wired environment, has a vague and ever evolving notion of perimeter. This vague perimeter makes the security of the system a challenge due to ability to add access points (known as rogue access points) that are unknown to the system administrator. Further, the wireless environment permits attackers to intercept data in transit easier than in the wired environment. To combat this, the data suffers the overhead penalty of encryption. The encryption solution itself is not a true answer to data protection because it also has vulnerabilities which introduce the wireless computing security paradox that will not be further discussed. Lastly, mobile devices are somewhat limited in their computing power (though this is rapidly changing).

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROPOSED SOLUTION

A. INTRODUCTION

In this chapter, a database (DB) application architecture capable of bridging the conceptual gap between object-oriented software development technologies and Relational Database Management Systems (RDMBSs) discussed in Chapter I will be proposed. This proposed DB architecture will be arrived at by first making decisions on the key elements discussed in Chapter II in the context of U.S. military DB applications. Decisions will be provided in the following areas: (1) DB Application Software Architecture, (2) Backend DBMS, (3) Client Connectivity, (4) Database View, and (5) Application Logic Architecture. Based on each of these decisions an overall architecture will be proposed providing a basis for Chapter IV's proof of concept implementation.

B. DECISION ON DB APPLICATION SOFTWARE ARCHITECTURE

The current U.S. military environment demands three things from DB applications: (1) high scalability (support for several end users), (2) support for several DBs within the application, and (3) provide capability to utilize mobile devices. Each of the three DB application software architectures introduced in Chapter II will be discussed in regards to these three demands.

1. One-Tier

The one-tier DB application software architecture, as discussed in Chapter II, does not scale well by definition. The one-tier DB application software architecture will support simultaneous access to multiple DBs. To support several DBs the one-tier DB application software architecture becomes very complex due to requiring an interface of some kind with each individual DB. This complexity makes software maintenance difficult in the case of multiple DBs and can also make adding more DBs in the future difficult. Furthermore, the single-tier DB application software architecture is not feasible in regards to mobile devices. Current mobile devices are limited in memory, secondary storage capacity, and processing power. Additionally, most DBs used by the military tend to be rather large. These mobile device limitations combined with the use of large DBs make the one-tier DB application software architecture infeasible.

If the mentioned mobile device limitations are overcome at some point in the future there is still an incompatibility with regards to DBs and the one-tier architecture for multiple users. This incompatibility is in the arena of data change management. Data change management is encountered when a user updates the data on his/her application, yet the rest of the users fail to get the change in a timely manner resulting in inconsistent information among users. The concern of data change management is an unavoidable deterrence to scalability, and as already mentioned prevents the one-tier architecture from being compatible in the military environment.

2. Two-Tier

Unlike the one-tier architecture, the two-tier DB application software architecture scales to many users and supports several DBs. Though it scales to many users, it fails to do so, well, in all cases. The two-tier architecture does not scale well in the case of a large number of users and databases in the same application. In the two-tier architecture if there are multiple users (M) and many DBs (N), then there are potentially M*N sessions open on the server. This M*N relationship is a potential bottleneck in the two-tier architecture and is of concern as two of the three requirements for a military DB application are support for multiple users and multiple DBs. Additionally, the two-tier architecture will support mobile devices, however; this support is limited in the military environment. The already discussed mobile device limitations combined with the increasing demand for mobile device applications in the military potentially over-tax the mobile device. This over taxation of mobile devices has a potential to restrict the relevance of current mobile devices to the military.

The two-tier DB application software architecture is a viable option in the military DB application because it is scalable, supports multiple users, and supports mobile devices. However, the two-tier architecture still has limitations in a military environment; specifically, the potential bottleneck and potential for application overload on mobile devices. These potential limitations make the two-tier architecture feasible yet still present undesirable consequences.

3. Three-Tier

The three-tier DB application software architecture is scalable and capable of supporting many DBs. The scalability of the three-tier architecture does not incur a bottleneck because this architecture does not have its application logic and data access combined at one layer. The three-tier architecture separates each of these layers allowing additional hardware to be placed at any potential bottlenecks, however; the three-tier architecture is more difficult to develop. The separation of layers decreases the burden on any one layer and allows more efficient access. Additionally, the three-tier architecture will support many DBs for this same reason. Furthermore, the three-tier architecture is capable of supporting mobile devices, however; the same concerns apply to mobile devices in three-tier architectures as those for the two-tier architecture.

The three-tier architecture's logical separation of functionality, as described in Chapter II, and its ability to support the necessary demands in a military environment make it a robust and capable architecture. Thus, since the three-tier architecture is highly scalable, capable of supporting several DBs, and mobile device capable it is the clear solution in military DB applications.

C. DECISION ON BACKEND DBMS

This section will provide an analysis of each of the DBMSs discussed in Chapter II arriving at a decision on what type of DBMS is the logical choice in a military environment. The current military environment requires three distinct qualities from a DBMS; product availability, product support, and low data migration cost (from the RDBMSs that currently exist in the vast majority of military DB applications). These three qualities will be considered for each type of DBMS.

1. Object-Oriented Database Management System (OODBMS)

Concerning product availability, the OODBMS is the least available of all DBMSs discussed in Chapter II. This is due largely in part to it being a relatively new DB technology. As with any new technology there is reluctance on the part of the software development industry to embrace such technologies. This reluctance limits the demand for the technology and therefore the number of vendors and related product support.

Migrating, or converting, the heavily used RDBMS data to OODBMS data is costly. First, the cost of training system administrators in the military, or any other size comparable organization, is a large investment and should be heavily weighted when considering converting to new technologies such as the OODBMS. Additionally, migration costs are compounded by the fact that the data has to be converted from a relational format to an object-oriented format. This conversion incurs not only a monetary cost but a time cost as well. The time cost is not limited to only the DB data. In most cases the DB applications that rely upon the DBMS must be converted as well in order to recognize data from an OODBMS vice a RDBMS. Thus, the lack of product support and product availability in conjunction with the high data migration cost make the OODBMS unsuitable for the military environment.

2. ORDBMS

The ORDBMS is gaining momentum in industry as RDBMS vendors see it as a way of combining concepts from the OODBMS and the RDBMS as discussed in Chapter II. This increase in momentum is in part due to the SQL like standard that is supported in the ORDBMS combined with its much desired ability to handle the modern object. However, the vast majority of the DBMS market is relational in nature and dominated by the RDBMS leaving the ORDBMS lagging in terms of product support and product availability, though; more supported and available than the OODBMS.

The data migration cost of converting RDBMS data to ORDBMS data is not as high as that of the OODBMS, yet it is still of concern. The ORDBMS is capable of storing primitive data types and therefore does not incur the time cost of converting all of its data, however; the data still must be written or copied into the ORDBMS from the original military RDBMS. Like in the case of OODBMS, the cost of training administrators to use the new technology must be considered, though the cost is not as drastic when compared to OODBMSs. Further, if the applications that interact with the ORDBMS are to be fully capable of utilizing ORDBMS features, such as object compatibility, then they too must be updated. This high data migration cost combined with the lack of product availability and product support make the ORDBMS an inappropriate choice for military DB applications.

3. RDBMS

The RDBMS is the dominant DBMS on the market today, as discussed in Chapter I, and the standard to which other DBMSs are compared. This dominance combined with the longstanding use and availability of the RDBMS provides for a widely available product from many vendors that is heavily supported in industry. Further, this dominance is quite clear in the military where nearly all DB applications are reliant upon the RDBMS. Additionally, the military already has existing relationships with various RDBMS vendors regarding support for their respective products.

The data migration cost in the case of RDBMSs in the military is nonexistent in most cases as the data already resides in an RDBMS. However, if there is an upgraded RDBMS available then the time of migrating data from the old RDBMS to the new is of some concern, yet unavoidable. Therefore, the high availability and support in conjunction with the low to nonexistent data migration cost establish the RDBMS as the logical choice for DBMSs in military DB applications.

D. DECISION ON CLIENT CONNECTIVITY

As discussed in Chapter II, both wired and wireless environments have their merits and shortcomings. The military is increasingly becoming mobile device capable regardless of any shortcomings. This increased use of mobile devices is driving the military toward a wireless capable environment in order to support the high demand for remote access. The compelling force toward a wireless environment in the military is ease of access to information. A key example of this in the military is in the case of a Duty Officer. Each branch of the military has some form of a Duty Officer who is on call and traditionally carried a pager as a means of contact. As time and technology progressed, the pager eventually gave way to the cell phone. Further, as the lines between cell phone technologies and other handheld device technologies (e.g. PDAs) become increasingly blurred so does the Duty Officer’s ability to access data remotely. The military sees that the Duty officer having a means to reach out to any relevant source of data via his mobile handheld device as an emerging necessity.

The desire for a wireless DB environment in the military is driven by three primary factors. First, the mobile devices are already present and in use as a part of every day military operations. Second, the mobile devices require wireless access in order to provide time relevant information to the user. Third, the demand for time relevant information to where it is needed is a growing requirement. These three factors are key enablers to military personnel because they significantly enhance productivity and the ability to make sound decisions.

E. DECISION ON DATABASE VIEW

The modern object-oriented approach to software development provides software developers with all the benefits of Object-Oriented Programming Languages (OOPLs) as discussed in Chapter I. These benefits allow the developers to use objects to closely model real world items. Further, allowing the software developers to view data in the DB as objects decreases the work required by the software development professional. This decrease in work is derived from the lack of data conversion. Data conversion is not required prior to being used by the developer because the data is already in object format. In the case of the military which is heavily reliant upon the RDBMS, it is essential that an object-oriented view of the DBMS be provided to the software developers in order to efficiently make use of modern software development technologies.

From a user's perspective, by providing the software developers an object-oriented view of the DB it allows for a more rapid development of applications providing useful software in a more timely manner. Additionally, by providing the user an object-oriented view of the backend relational DB they are not required to be familiar with the relational DB operations (such as table joins and queries) and structures. Specifically, this object-oriented view of the database provides the user with a more intuitive means of interacting with the information that is not restricted to the relational format consisting of tables with rows and columns.

An object-oriented view of backend databases is beneficial to both the user and the developer as previously discussed. These benefits also carry over to the military in general. By providing an object-oriented view of its backend databases the military can incorporate modern software into its existing DB applications with ease. Further, this can lead to savings in the form of both development time and cost. These savings and benefits could potentially enhance military DB services in a broad range of areas to include payrolls, muster, combat information, and pension records.

F. DISCUSSION ON APPLICATION LOGIC ARCHITECTURE

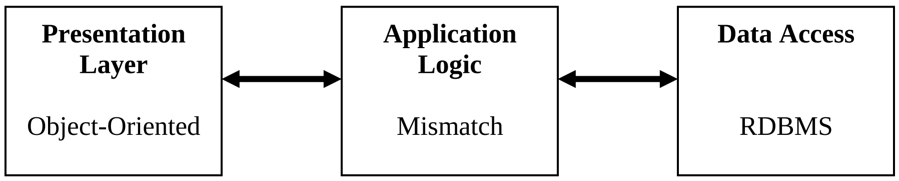


Figure 9. Object-Relational Mismatch in a Three-Tier Architecture

As previously concluded, a three-tier DB application is best suited in a military environment. Additionally, the RDBMS is the logical choice of implementing the data access layer of that three-tier architecture, yet an object-oriented view is required at the presentation layer. Using a RDBMS and requiring an object-oriented view of the data provides a conceptual mismatch, known as the Object-Relational Mismatch (or Impedance Mismatch). The basic three-tier architecture can be seen in Figure 9 that captures where that mismatch occurs and must be addressed. This section will briefly discuss that mismatch and conclude with a means of overcoming that mismatch.

The Object-Relational (OR) Mismatch, or Impedance Mismatch, is encountered when the relational paradigm meets the object-oriented paradigm or vice versa. This mismatch is formed by the relational paradigm being founded by mathematical principles whereas the object-oriented paradigm is founded by software engineering principles (Scott, 2006). The differences between the founding principles of the paradigms lead to the mismatch. The relational paradigm is based on storing data in tables consisting of columns and rows and is retrieved via Structured Query Language (SQL). Further, the relational paradigm represents relationships among data stored in those tables by joining tables. In contrast, the object-oriented paradigm is based on storing data and the data's associated behavior in the form of objects representing the relationships among data via the objects themselves. Though these two paradigms when combined cause the OR mismatch, individually they provide indisputable advantages as discussed in Chapter II. In order to make use of the individual advantages this mismatch must be overcome.

This OR mismatch is unavoidable in the proposed architecture shown in Figure 9. The Application Logic Layer is the clear location to handle this mismatch. There are two apparent solutions to overcoming this mismatch. First, the software developer may create their own methods of converting the objects from the presentation layer to a relational data format in order to persist them in the RDBMS. Additionally, the developer must then create their own methods of converting the relational representation of that data to an object acceptable by the presentation layer. The choice of creating methods to handle this mismatch can become tedious and time consuming in the case of large applications and databases. Second, the software developers may use existing software that is designed to handle the OR mismatch, an OR Mapper. By using an OR Mapper the Application Logic Layer essentially provides a virtual OODBMS to the presentation layer. This allows the object-oriented Presentation Layer to communicate with the Data Access Layer via the Application Logic Layer overcoming the OR Mismatch.

G. SUMMARY

In this chapter DB application design decisions were made in five specific areas in an attempt to bridge the gap between modern object-oriented software development technologies and the heavily used RDBMS. Again, these five decision areas are: (1) DB Application Software Architecture, (2) Backend DBMS, (3) Client Connectivity, (4) Database View, and (5) Application Logic Architecture. These decisions provide a basis for developing a prototype DB application in the military.

The decision on DB application software architecture analyzed the three prominent architectures in a military context and arrived at a logical decision. This decision was based upon the architecture being capable of meeting the military demands of scalability, support for multiple DBs, and being mobile device capable. The analysis concluded with the three-tier DB application software architecture being the optimum solution for the military environment.

The Backend DBMS decision analyzed the three dominant DBMSs available and selected the DBMS that was the best fit. These DBMSs were contrasted by their product availability, their product support, and the cost of data migration. Upon conclusion of this analysis the DBMS that was best suited for military use was the RDBMS. The

RDBMS presented the highest product support and availability while incurring the lowest data migration cost making it the natural choice.

The next required decision was regarding the means of client connectivity. This discussion presented the requirement for mobile device access. The mobile device access in turn mandates wireless connectivity for the client. The requirement for mobile devices was founded by the demand for remote access to time relevant information and the large availability of such devices.

Additionally, a discussion on database view was provided. Here the decision was arrived at that an object-oriented view was required. This requirement was justified by the benefits provided to the software developer, the user, and the military in general. These benefits, as stated previously, were enabled by the use of an OOPL.

Finally, the decision to use a RDBMS and provide an object-oriented view of the Database presented the object-relational mismatch. Further, the clear location to address this mismatch was in the middle layer of the three-tier DB application software architecture, the Application Logic Layer. The OR Mapper was the chosen method of addressing this mismatch.

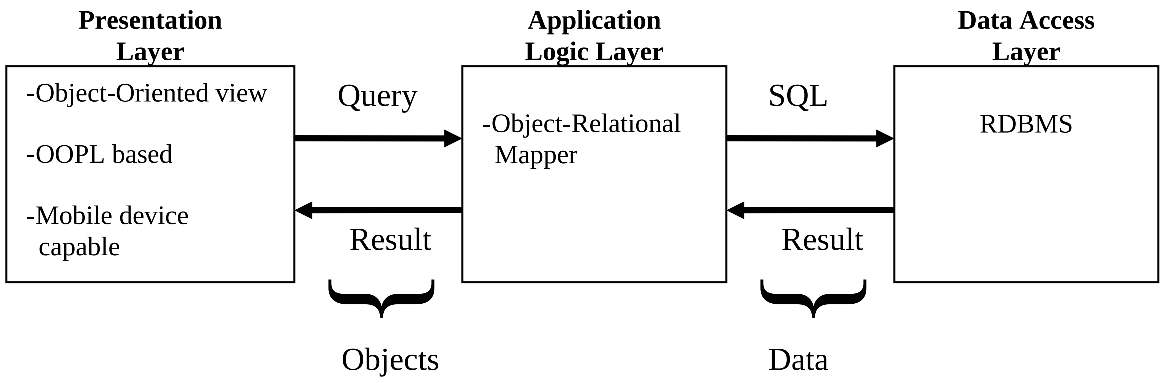


Figure 10. Proposed Three-Tier DB Application Architecture

Based upon the design decisions presented, the proposed three-tier DB application architecture (Figure 10) was devised. This three-tier DB application software architecture is logically separated into the three layers as shown in Figure 10. The presentation Layer maintains the OO view of the DB by using an OOPL while being mobile device capable. Additionally, the presentation layer interacts with the application logic layer by passing an object that captures the user's request (query) to the application logic layer and receives results in object form. In turn, the application logic layer converts the user's query object to a SQL based query via the OR Mapper. Conversely, the OR Mapper receives the results of the query from the data access layer and converts that data to object format. Lastly, the data access layer uses a RDBMS. This proposed solution provides a means of bridging the conceptual gap between the object-oriented paradigm and the relational paradigm while meeting modern military DB application requirements.

IV. IMPLEMENTATION

A. INTRODUCTION

This chapter will provide a proof-of-concept prototype database (DB) application that shows the viability of the proposed architecture devised in Chapter III. This prototype will provide a means to overcome the Object-Relational (OR) Mismatch while allowing mobile device access. The prototype will be presented by first discussing the application domain followed by a detailed discussion of the prototype's three-tier architecture. The chapter then concludes with a description of the prototype's design architecture implementation, a sample interaction, and findings during implementation.

B. APPLICATION DOMAIN

The prototype DB application will focus on the commonly used personnel DB application utilizezd by all military branches. Specifically, the prototype will present a Joint Staff personnel DB application that allows mobile device access. Further, the prototype application would be used to gain rapid access to personnel information. This personnel information would then be used to provide for timely reports. For example, the prototype DB application could be utilized by systems such as the Personnel Casualty Report (PCR) System used by the U.S. Navy (USN) and U.S. Marine Corps (USMC). A (PCR) is an electronic message containing casualty information for the purpose of reporting as well as a source of information used to inform the next of kin of a casualty status. Overall, the prototype will allow real time access to administrative information and provide a more intuitive means of representing the information in USN and USMC DB applications to non-expert users. Here a non-expert user is defined as a user that has no knowledge of DB functionality and design or how to retrieve data from the DB directly, for example, using Structured Query Language (SQL) based queries.

C. PROTOTYPE'S THREE-TIER ARCHITECTURE

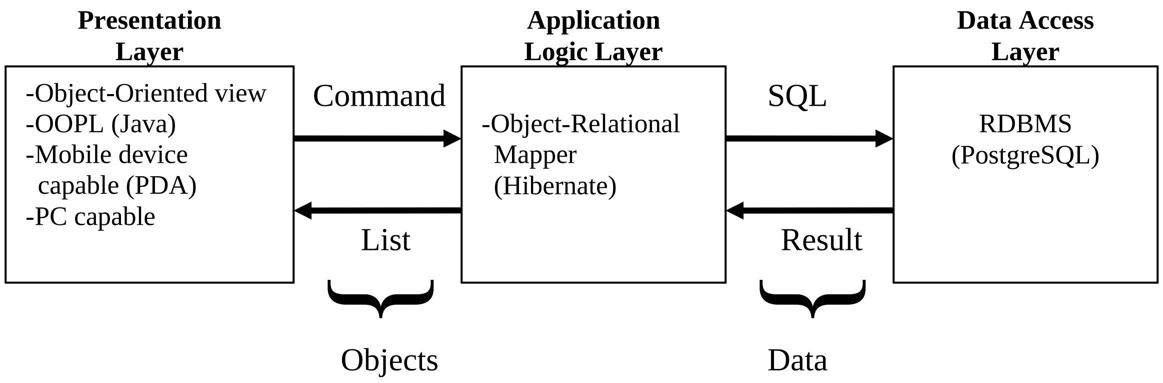


Figure 11. Prototype Three-Tier DB Application Architecture

The prototype's architecture, as seen in Figure 11, is logically separated into three layers. These layers are the data access layer, the application layer, and the presentation layer. Each of these layers will be described in more detail to include relevant technologies (Java, PDA, Hibernate, and PostgreSQL) used to implement design decisions arrived at in Chapter III. Additionally, the interaction between each layer will be presented. Lastly, each layer will be described with regards to the military application domain as previously described.

1. Data Access Layer

The goal of the data access layer is to provide a means of data storage using a Relational Database Management System (RDBMS) as described in Chapter III. The prototype uses a RDBMS to capture the personnel information and relationships for a Joint Staff.

The RDBMS for this prototype was chosen based on four criteria. First, the RDMBS must be open source due to thesis funding constraints. Second, the RDMBS must provide standard SQL as a means of data access because SQL is common to military RDBMSs. Third, the RDBMS must provide a means of data modification other than SQL for ease of inserting/removing data to facilitate both trouble shooting and application testing. Lastly, the RDBMS must be Java Database Connector (JDBC) capable in order to interface with Java applications. Based on meeting the selection criteria the specific RDBMS chosen for this prototype was postgreSQL, specifically version 8.0.

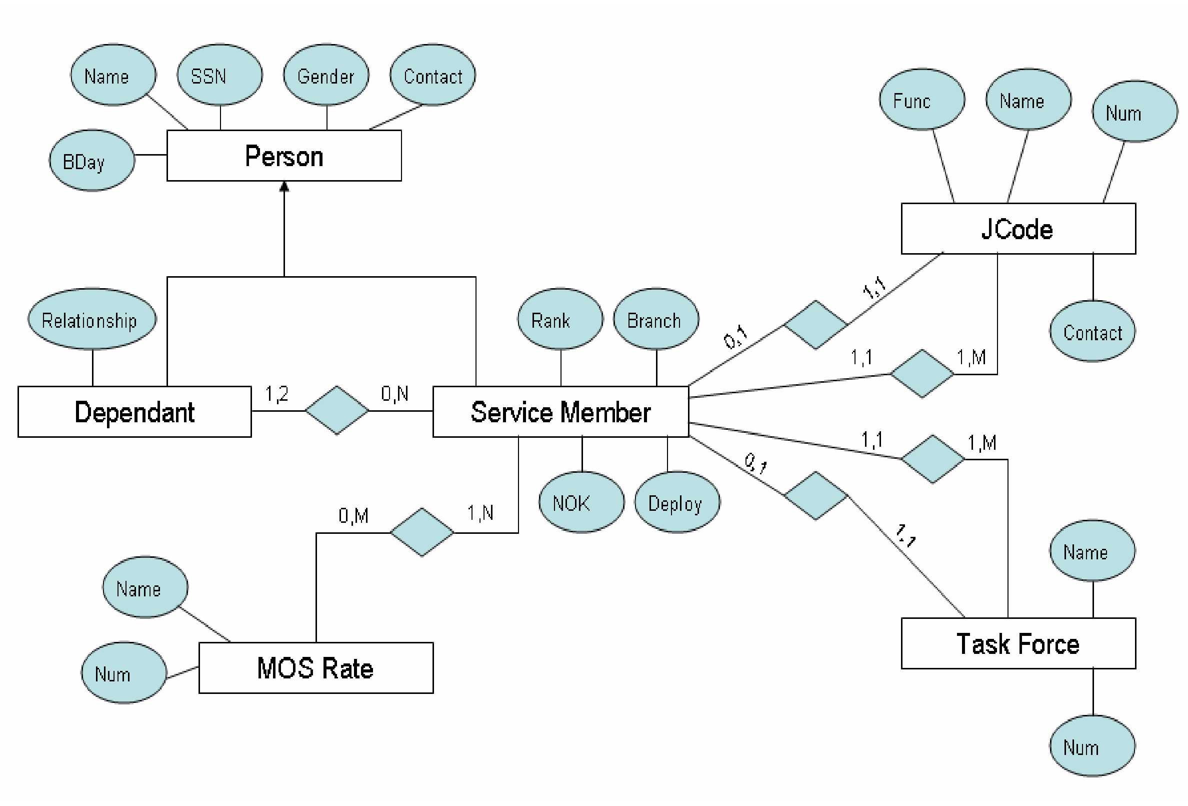


Figure 12. Prototype Entity Relationship Diagram

As seen in Figure 12, the information stored in the RDBMS closely models that of a real Joint Staff. However, none of the information used was actual military personnel information in keeping with the Privacy Act of 1974. Additionally, as seen in the Entity Relationship (ER) Diagram (Figure 12), the RDBMS captured all possible relationships between tables. Specifically, there are instances of one-to-one, one-to-many, and many-to-many bi-directional relationships. Further, there is an instance of an inheritance relationship in the ER diagram. The inheritance relationship could be represented via one-to-one relationships between entities, however; an object-oriented view of the DBMS is desired. Inheritance is an object-oriented paradigm concept and does not necessarily have a counterpart in the relational paradigm, yet since an object-oriented view of the DBMS is essential to the scope of the thesis it was included in the schema for implementation.

As previously stated, the ER diagram (Figure 12) represents the personnel information of a Joint Staff. A Joint Staff is comprised of JCodes, analogous to a department of a corporation. Further, the Joint Staff can be segmented to form a Task Force that carries out specific functions under the purview of either the Joint Staff or the Joint Staff's higher command. Joint Staff personnel are represented in the ER diagram as instances of people in the Person table (where each row in the table represent a person). These people can then be further categorized as either Service Members or Dependents, thus the inheritance relationship between the Person, Service Member, and Dependent tables. These Dependents and Service Members are related in a one-to-many relationship where a Dependent can be related to one or two Service Member entities. Conversely, the Service Member Entities can be related to either multiple or no Dependent entities. Additionally, the Service Members can have multiple Military Occupational Specialties (MOSs) or Rates (analogous to a job title or specialty). The relationship between the Service Member and the MOS Rate tables is represented in the ER diagram as a many-to-many relationship where the Service Member entities can be related to multiple MOS or Rate entities. Further, the Service Member to MOS Rate relationship is also bidirectional. Additionally, the Service Member entities are further related to a JCode entity by dual relationships. First, the Service Member entities have a bi-directional one-to-one relationship with the JCode entity that represents the Service Member in charge of each JCode. Second, the Service Member entities have a bi-directional one-to-many relationship with the JCode entities as each Service Member entity can be related to only one JCode entity. Furthermore, the relationships between the Service Member and the Task Force entities are identical to those between the Service Member and JCode entities.

The ER diagram was implemented in the PostgreSQL RDBMS. The initial method of creating the DB was to use a GUI based DB design tool, specifically DB Designer 4.0. DB Designer allowed for easy modeling of the ER diagram in PostgreSQL without the tedious process of manually creating the appropriate SQL script. Further, DB Designer allowed for easy modification of the schema in order to capture all relationships in the Joint Staff appropriately. Of note, this method of creating the DB in PostgreSQL was only used to initially set up the schema and was later replaced by using the Object Relational (OR) Mapper tool as described in more detail later.

The data access layer as seen in the Prototype Three-Tier DB Application Architecture diagram (Figure 11) will directly interface with the application logic layer. Specifically, the data access layer will receive a SQL query from application logic layer and return the resulting data. The interaction between layers will be enabled by a JDBC connection between the application logic layer and the data access layer.

2. Application Logic Layer

The application logic layer as stated in Chapter III, will overcome the OR Mismatch between the data access layer and the presentation layer. The means to overcome this mismatch is via an OR Mapper. Additionally, as stated in previous chapters, the application must allow for wireless connectivity due to the growing military demand. This section will discuss the choice of a specific OR Mapper, specific means of wireless connectivity, and the incorporation of those items into the three-tier architecture at the application logic layer.

The specific OR Mapper chosen for this prototype must meet four minimal criteria; (1) the mapper must be configurable to any RDBMS, (2) the mapper must provide sufficient support and availability, (3) the mapper must be open source, and (4) the mapper must be capable of interfacing with a JDBC connection. Based on these criteria and available OR Mappers, the specific OR Mapper chosen for implementation was Hibernate (more specifically, version 3.1). Hibernate is configurable, has sufficient support, is open source, and is capable of interfacing with a JDBC connection making it a practical solution for the prototype’s three-tier application architecture.

As discussed in Chapter III, it was necessary to allow for both wired and wireless connectivity to the application. The method of allowing for this dual connectivity was by using a connection manager capable of handling multiple clients of both wired and wireless configurations. Specifically, Apache Tomcat version 5.5 was chosen for these capabilities. Further, Apache is a supported by Hibernate as a means of connection pooling and is also open source.

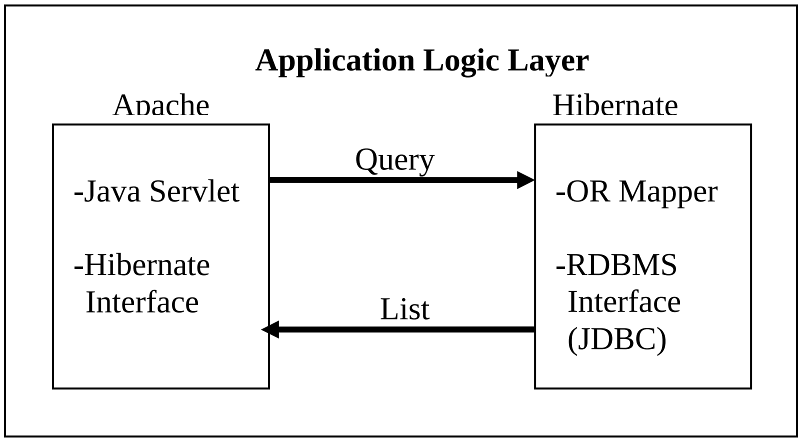


Figure 13. Application Logic Layer Implementation

As seen in Figure 13 above the application logic layer utilizes Hibernate and Apache to overcome both the OR mismatch and provide for wired and wireless connectivity. The Apache Servlet will receive a query in object form from the presentation layer and pass that query object to the Hibernate Interface where the query is converted to a query recognized by Hibernate. Hibernate then executes the query via its JDBC interface with the data access layer and converts the query result to a list of java objects. That list of objects is passed back to the Apache Servlet where it is transmitted to the presentation layer.

3. Presentation Layer

The presentation layer's goals during implementation were threefold; (1) to be object-oriented, (2) to provide the users access to a backend DBMS via both a mobile device and a PC, and (3) to provide an effective and easy to use querying tool for non-expert users. These goals comprise the overall objective of providing a more intuitive means of representing data for military DB applications in a mobile environment. The means of achieving each of these goals and appropriate implementation decisions will be further discussed.

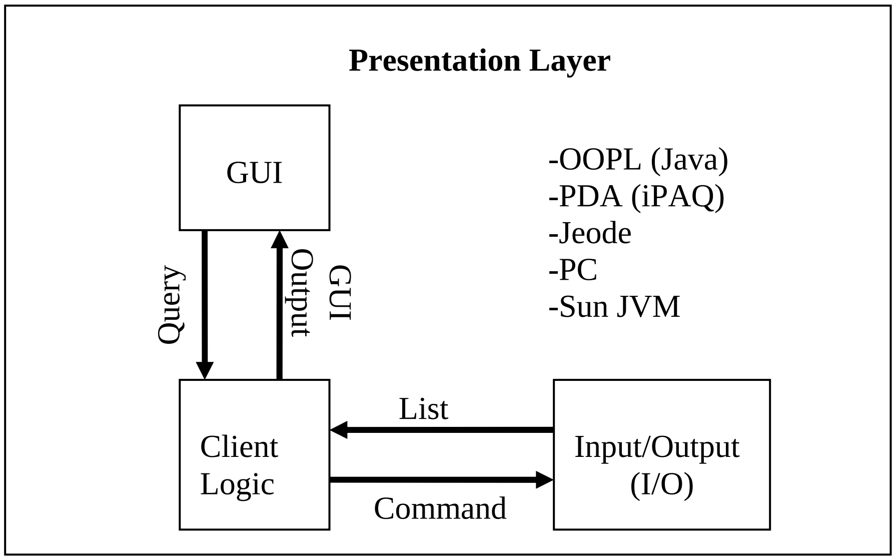


Figure 14. Presentation Layer Implementation

The presentation Layer of the DB application, as seen in Figure 14, presents the Graphical User Interface (GUI) to the user. As discussed in Chapter II, this GUI can be of three primary formats; the Pure Web Browser, the Java Applet, and the Full Software Application. The Full Software Application was chosen for its ability to provide greater flexibility in GUI design and the ability to provide user defined data access to the DBMS data. Further, an Object Oriented Programming Language (OOPL) was a requirement as stated in previous chapters. The prototype DB application utilizes Java as its OOPL due to Java being the current industry standard, extremely portable, and well supported. Additionally, Java is supported for mobile device application (specifically the PDA) development meeting the mobile access requirement as discussed in Chapter III.

Further, Java allows the same GUI, or program code, to be executed on both the PDA Client and the PC. The prototype accomplishes this by utilizing Jeode, a Java Virtual Machine (JVM), for the PDA application and a Sun JVM on the PC. Writing one program supported on both platforms requires that the presentation layer GUI be written to the more restrictive Java libraries of Jeode. Specifically, the application must be compiled to Java Developer Kit (JDK) version 1.1 or older. Additionally, the GUI design centers on providing the user a proper display for both the PC and the PDA while maintaining the same java program code for both platforms. The specific PDA chosen for this application was the Hewlett Packard (HP) iPAQ 5500. This model of PDA is similar to the rugged version of the PDA being fielded by the USMC, the Dismounted Data Automated Communications Terminal (D-DACT).

As seen in the Presentation Layer Implementation diagram (Figure 14), the presentation layer is logically segmented into the GUI, the Client Logic, and the Input/Output (I/O) segments. These segments break up the overall functionality of the presentation layer into logical partitions. The presentation layer begins when the user inputting the selection criteria into the GUI forming the initial query to include the requested type of object to be returned. The user query is passed to the Client Logic where it is converted to a Command Object and then passed on to the I/O segment of the presentation layer. The I/O segment will then transmit the Command object to the Apache Servlet and conversely receive a list of objects from the Apache Servlet that match the selection criteria initially requested by the user. The returning result list is passed to the Client Logic segment where it is iterated through based on the object types contained in the list. Those objects are then converted to the proper format and presented to the GUI for display to the user.

D. PROTOTYPE DESIGN AND ARCHITECTURE

This section will advance the architecture presented in the Prototype Three-Tier DB Application Architecture diagram (Figure 11) to a realization of the prototype DB application. The prototype design and architecture discussion will include key program code and methods that are critical to understanding the prototype DB application. Additionally, Unified Modeling Language (UML) diagrams will be provided for the presentation and application logic layers. These UML diagrams will be discussed in detail in order to supply a visual representation of the DB application Java class relationships. The prototype DB application will serve as a proof-of-concept that will provide handheld device access to a Joint Staff personnel DB demonstrating the viability of the architecture proposed by this thesis.

1. Overall Design

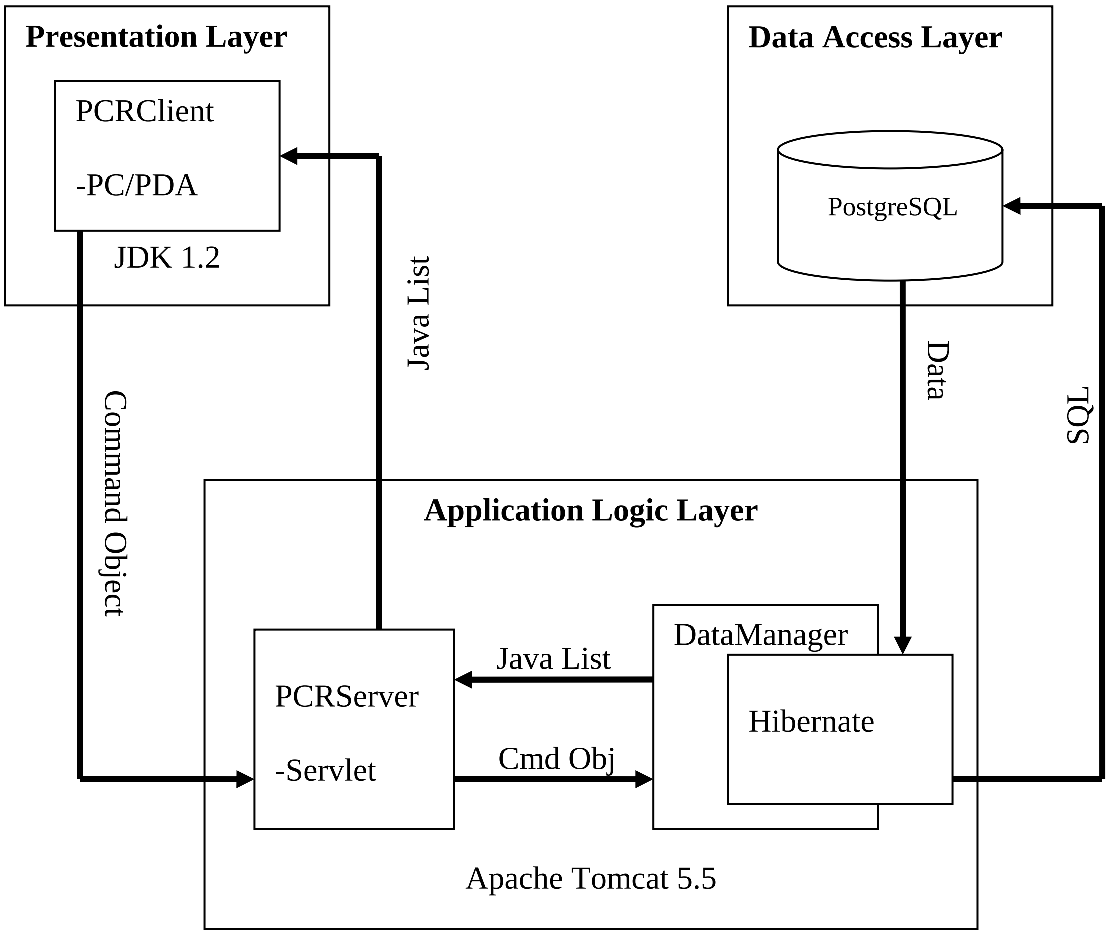


Figure 15. Implemented Prototype DB Application Architecture

As seen in Figure 15 above, the prototype DB application is represented in a three-tier form. Further, these tiers (or layers), though logically separated, interact with one another to achieve the overall goals presented by the thesis. This interaction between layers will be described as a request for information is initiated and then that request will be followed through the DB application architecture where it is processed and the results are displayed to the user. Later in the chapter a sample interaction will be provided and this process will be repeated for a specific instance of a request.

Initially, the user interfaces with the PCRClient program in order to request information regarding the Joint Battle Staff's personnel DB. This request is captured in the PCRClient as a Command object and is transmitted over a Hypertext Transfer Protocol (HTTP) connection to the Apache Tomcat Servlet (PCRServer). The PCRServer in turn passes the Command object to an instance of the DataManager class. DataManager then interprets the Command object contents and generates a Hibernate Query Language (HQL) query. The HQL query is then converted to SQL by Hibernate and passed to PostgreSQL via the JDBC connection. PostgreSQL in turn executes the SQL query and returns the results to Hibernate. Hibernate then captures the returning data as a list of objects and presents that list to DataManager where it is handled appropriately (this will be discussed in more detail later). After DataManager processes the list it passes the results in a new list of objects to PCRServer. Lastly, PCRServer transmits the list to the PCRClient where it is processed and displayed to the user. Each of these layers and classes will be discussed in more detail in the following sections in order to provide a better understanding of the processes that take place at each layer and within each class.

2. Data Access Layer

The data access layer consists of the RDBMS, PostgreSQL 8.0, installed according to its documentation. This RDBMS captures the relationships of a Joint Staff personnel DB as seen in the ER diagram (Figure 12). Further, this DB was initially created using DB Designer as already discussed, however; Hibernate was used to create the final version of the DB. The DB table structure and contents will be described for a specific table, Service Member. The remaining tables of the DB are similar in scope to Service Member therefore no additional explanation will be provided.

Understanding the structure of the Service Member table is critical to understanding the relationships that exist in the DB because the Service Member table is the hub of the DB. The structure of this table as represented by the PostgreSQL Admin Tool in SQL form is shown below:

```
CREATE TABLE sm
(
  per_id int8 NOT NULL,
  rank varchar(255),
  branch varchar(255),
  nok varchar(255),
  deployed bool,
  jcode_id int8,
  tf_id int8,
  CONSTRAINT sm_pkey PRIMARY KEY (per_id),
  CONSTRAINT fke5a2acdb92a FOREIGN KEY (per_id) REFERENCES per (per_id) ON UPDATE
             NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fke5a679e6502 FOREIGN KEY (jcode_id) REFERENCES jcode (jcode_id) ON
             UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fke5ae6fef916 FOREIGN KEY (tf_id) REFERENCES tf (tf_id) ON UPDATE NO
             ACTION ON DELETE NO ACTION
)
WITH OIDS;
```

This SQL representation of the table shows the table constraints. Further, these constraints were generated by Hibernate upon DB creation. The rational for these table constraints will be provided later in the Hibernate discussion. As seen in the SQL above, the primary key of the Service Member table is the per_id column. Furthermore, this column is used to capture the relationship between the Service Member table and all other tables in the DB by acting as a foreign key as appropriate.

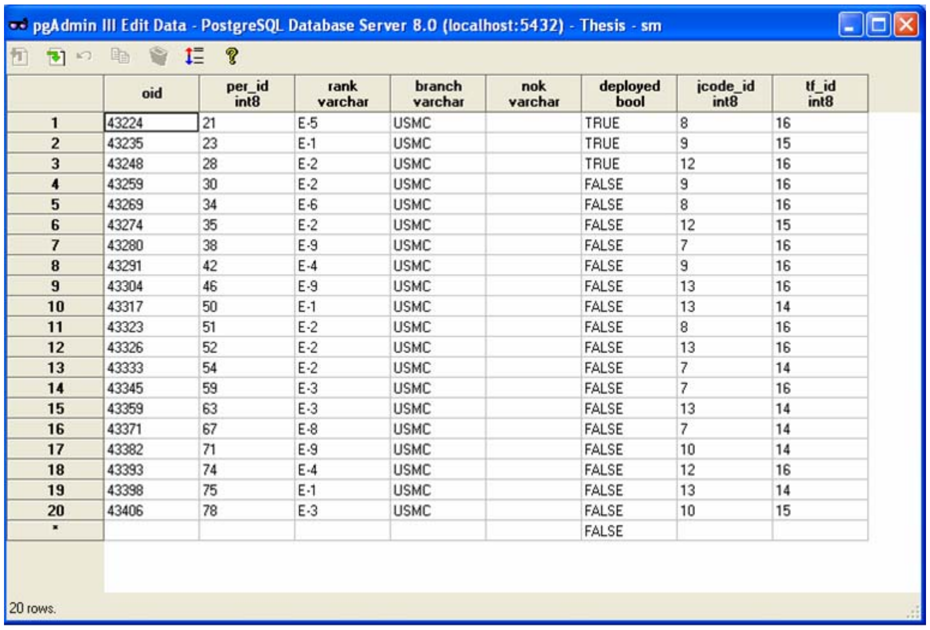
Beyond creating the tables in the DB there needed to be information representative of personnel data for a Joint Staff. This need was met by using the java DataFiller class. Vital portions of the DataFiller class are provided below with an explanation of their purpose and functionality. The main method of this class is shown below:

```
public static void main(String[] args) {

        if(args[0].equals("fillDB")){
                mgr.fillDB( );
        }

        HibernateUtil.getSessionFactory().close();
}
```

The main method simply calls the fillDB( ) method in order to populate the DB by using Hibernate. After the fillDB( ) method has populated the DB the session that Hibernate has open with the DB is closed. The fillDB( ) method is provided below followed by an explanation of how it populated the tables of the DB with information representative of a Joint Staff personnel DB:

```
public void fillDB( ){

        //create and save MOSs, Jcodes, TFs to DB
        ...

        for(int i = 0; i < 20; i++){

                //get session from Hibernate
                Session s = HibernateUtil.getSessionFactory().getCurrentSession();
                //begin transaction
                s.beginTransaction();


                //create ServiceMember & Depedent objects to be saved to DB
                ServiceMember sm = mgr.createSM(i);

                //create a random number of dependents per servicemember up to 4
                int numDep = getRandom(5);

                for(int j = 0; j < numDep; j++){


                        Dependent dep = mgr.createDep(j, sm);
                        s.save(dep);
                }

                //Add up to 4 MOS's to each ServiceMember
                int numMOS = getRandom(4);

                for(int j = 0; j < numMOS; j++){

                        int mosNum = getRandom(mosList.length);
                        sm.addToMOS(mosList[mosNum]);
                        s.update(mosList[mosNum]);
                }

                //Assign each ServiceMember to a Jcode
                int jcodeNum = getRandom(jcodes.length);
                jcodes[jcodeNum].addToServiceMembers(sm);
                s.update(jcodes[jcodeNum]);

                //Assign each ServiceMember to a TaskForce
                int tfNum = getRandom(tfs.length);
                tfs[tfNum].addToServiceMembers(sm);
                s.update(tfs[tfNum]);

                s.save(sm);

                //commit the transaction to the database
                s.getTransaction().commit();
        }

}
```

Initially, the method fills the entries for the MOS, Jcode, and TaskForce tables. The remaining tables are then filled by the main loop of the method, specifically twenty Service Member table entries. The main loop begins by getting a session from Hibernate and beginning a transaction with PostgreSQL. ServiceMember objects are then created and given a random number of Dependent objects not to exceed four. This is followed by an assignment of MOS(s), Jcode, and TaskForce objects to each ServiceMember which are then persisted to PostgreSQL via Hibernate by committing the transaction. Table 1 below shows the resulting relation.


pgAdmin III Edit Data - PostgreSQL Database Server 8.0 (localhost:5432) - Thesis - sm

	oid	per_id int8	rank varchar	branch varchar	nok varchar	deployed bool	jcode_id int8	tf_id int8
1	43224	21	E-5	USMC		TRUE	8	16
2	43235	23	E-1	USMC		TRUE	9	15
3	43248	28	E-2	USMC		TRUE	12	16
4	43259	30	E-2	USMC		FALSE	9	16
5	43269	34	E-6	USMC		FALSE	8	16
6	43274	35	E-2	USMC		FALSE	12	15
7	43280	38	E-9	USMC		FALSE	7	16
8	43291	42	E-4	USMC		FALSE	9	16
9	43304	46	E-9	USMC		FALSE	13	16
10	43317	50	E-1	USMC		FALSE	13	14
11	43323	51	E-2	USMC		FALSE	8	16
12	43326	52	E-2	USMC		FALSE	13	16
13	43333	54	E-2	USMC		FALSE	7	14
14	43345	59	E-3	USMC		FALSE	7	16
15	43359	63	E-3	USMC		FALSE	13	14
16	43371	67	E-8	USMC		FALSE	7	14
17	43382	71	E-9	USMC		FALSE	10	14
18	43393	74	E-4	USMC		FALSE	12	16
19	43398	75	E-1	USMC		FALSE	13	14
20	43406	78	E-3	USMC		FALSE	10	15
*						FALSE		

20 rows.

Table 1. Service Member Table in PostgreSQL

Table 1 represents the Java created ServiceMember objects from the DataFiller class in relational format. The other tables are similar in nature and correspond to the ER diagram of a Joint Staff. Furthermore, the ServiceMember class is a subclass of the Person class exercising inheritance in OOPL, specifically Java. Additionally, inheritance is modeled in the ER diagram and captured in the RDBMS as table constraints between the Person and ServiceMember tables. Thus, inheritance in the RDBMS is enabled via the OR Mapper.

3. Application Logic Layer

The application logic layer is where the majority of the implementation takes place. This layer, as seen in the Implementation Prototype DB Application Architecture diagram (Figure 15), consists of a PCRServer, a DataManager, and Hibernate. Overall, the application logic layer receives a user request for information in the form of a Command object and converts that object to a SQL query. That query is then executed via the interface with the data access layer and that query's results are then converted to a list of objects and returned to the presentation layer. This section will provide a detailed description of the PCRServer, DataManager, and Hibernate. This description will include a UML diagram and critical segments of program code in order to facilitate a better understanding of the application logic layer to the reader.

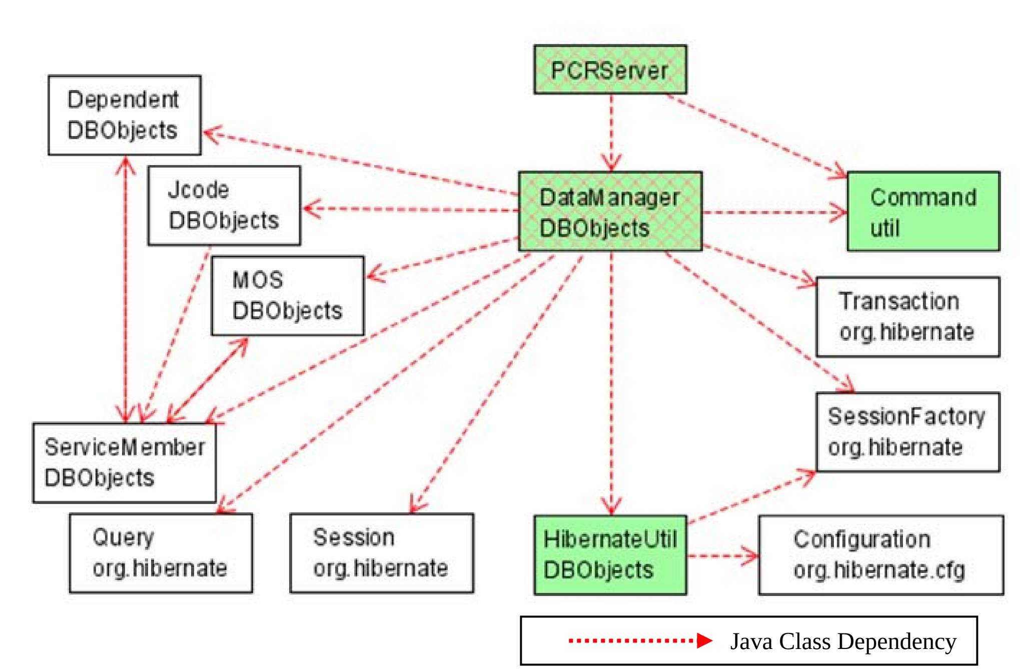


Figure 16. Application Logic Layer UML Diagram

The UML diagram above (Figure 16) represents the java class structure of the application logic layer. Here the primary java classes are shaded and will be discussed in more detail. The application logic layer begins by waiting for input from the presentation layer via the PCRServer class. This class is a Java HttpServlet that provides remote access via a Universal Resource Locator (URL) to a web server, specifically Apache Tomcat 5.5. Key portions of the PCRServer class are provided below:

```
public class PCRServer extends HttpServlet {
        ...
        public void doPost(HttpServletRequest request, HttpServletResponse
              response) throws IOException, ServletException {
                //create datamanager object
                DataManager mgr = new DataManager();


                //cmd items
                Command serverCommand = new Command();
                //Receive Obj from server
                ObjectInputStream in =

                        new ObjectInputStream(request.getInputStream());

                try{//rtv cmd object and set var

                        //get cmd obj

                        serverCommand = (Command) in.readObject();

                }catch (Exception e) {

                        System.out.println("Problem retrieving object");

                }

                in.close();


                //do work
                List outList = mgr.getResults(serverCommand);


                ObjectOutputStream out =
                        new ObjectOutputStream(response.getOutputStream());

                out.writeObject(outList);

                out.flush();
                out.close();
                }
            }
```

The PCRServer receives input from the PCRClient via the doPost( ) method. The doPost( ) method creates a DataManager and Command object and then opens an ObjectInputStream. The Command object and all other objects that are transmitted via the HTTP connection implement the java.io.Serializable interface. Further, all objects transmitted in the prototype DB application use a java.io.ObjectStream. Specifically, ObjectInputStream receives the Command object from the PCRClient and then closes the ObjectInputStream. PCRServer then passes the Command object to the DataManager object and receives a list of objects in return. The ObjectOutputStream is then used to send the list of objects to the PCRClient and is closed. The process presented by the PCRServer allows the Command object containing the user query to be sent to the DataManager class and receive the corresponding results from the DB as a list of objects.

DataManager receives the Command object containing the user query from PCRServer, converts the Command object to an HQL query and uses Hibernate to retrieve the requested information from PostgreSQL. The retrieved information is then returned to DataManager and ultimately PCRServer as a list of objects. DataManager will be discussed in more detail to include appropriate abbreviated segments of program code to facilitate an understanding of how this conversion process occurs. As shown previously the PCRServer passes the Command object to the DataManager’s getResults( ) method shown below:

```
public List getResults(Command cmd){
        ...
        //process updates here
        if(cmd.getCmdType() == UPDATE){

        ...
        //process queries here
        }else if(cmd.getCmdType() == QUERY){
                resultList = processQuery(getQuery(cmd));
                //process list_db here
        }else if(cmd.getCmdType() == LIST_DB){

        ...
        }
        ...
        return resultList;
}
```

As the method above shows, the Command object is checked to determine what kind of action is being requested by the user. In the case of an information request, it is a query type Command object (for the scope of this thesis the prototype DB application was a read only application and did not allow for updates or additional functionality). Once identified as a query type of object, the Command object is sent to the getQuery( ) method. The getQuery( ) method in turn converts the Command object’s specific requests to an SQL query and is then passed on to the processQuery( ) method. Here, a list of objects is returned to DataManager and ultimately returned to PCRServer.

The processQuery( ) method, as discussed, processes the query by interacting with Hibernate. The specifics of how this is accomplished are provided below:

```
public List processQuery(String query){

        //get session from HibernateUtil and begin a transaction
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        session.beginTransaction();

        //create a result list to put data into
        List result = new ArrayList(LIST_SIZE);

        //create another list in the case the original is
        //further filtered by Jcode for instance
        List newList = new ArrayList(LIST_SIZE);

        //save the list of resulting objects from the query to a list
        result = session.createQuery( query ).list();

        ...
        /* Here the resulting list is processed and the objects that match
           the return type are copied into newList */
        //commit the transaction to DB
        session.getTransaction().commit();

return getCopied(result);

}
```

The method begins by establishing a session with Hibernate by requesting the current session from HibernateUtil. A Hibernate session is a connection to the DBMS allowing data access operations by using a transaction. Once that transaction is established with the PostgreSQL the HQL query is then passed to the Hibernate session's createQuery( ) method resulting in a Hibernate Query object. This Query object is then processed by the session and the corresponding results are returned as a list of objects. The resulting list of objects is obtained from Hibernate and then sent to the getCopied ( ) method where the list is modified to only allow for JDK 1.1 functionality for the PCRClient.

Hibernate is the workhorse of the application logic layer. Hibernate not only persists the Java objects to the RDBMS, it also executes queries with the DBMS and converts the results to Java objects. Hibernate was installed and configured in accordance with its online documentation to allow for its interaction with PostgreSQL. The prototype DB application utilizes a Extensible Markup Language (XML) file as a means of passing configuration parameters to Hibernate, specifically hibernate.cfg.xml, portions of which are provided below:

```
...

<hibernate-configuration>
        <session-factory>

                <!-- Database connection settings -->
                <property name =
                    "connection.driver_class">org.postgresql.Driver</property>
                <property name = "connection.url">
                    jdbc:postgresql://localhost/Thesis</property>
                <property name = "connection.username">username</property>
                <property name = "connection.password">password</property>

                ...
                <!-- SQL dialect -->
                <property name =
                    "dialect">org.hibernate.dialect.PostgreSQLDialect</property>

                ...
                <!-- Drop and re-create the database schema on startup -->
                <property name="hbm2ddl.auto">create</property>

                <mapping resource="Person.hbm.xml"/>
                <mapping resource="MOS.hbm.xml"/>
                <mapping resource="Jcode.hbm.xml"/>
                <mapping resource="TaskForce.hbm.xml"/>

        </session-factory>
</hibernate-configuration>
```

The Hibernate configuration file above shows the DBMS connection settings that allow interaction with PostgreSQL. These DBMS settings include the appropriate JDBC driver, the username, the password, and the appropriate dialect of SQL required to access PostgreSQL. Further, the configuration file allows for the schema to be dropped and recreated each time it is accessed (whenever Hibernate attempts to access PostgreSQL). Hibernate references each of the mapping resources listed in the configuration file to generate the schema in the DB via the appropriate SQL dialect. The mapping resources for the prototype DB application are the Person, MOS, Jcode, and TaskForce XML files.

The XML mapping files listed in the configuration file of Hibernate correspond to Java classes and PostgreSQL tables. Further, the mapping files allow Hibernate to transition from Java (an OOPL) to PostgreSQL (an RDBMS) and overcome the OR Mismatch as described in Chapter III. Overcoming the OR Mismatch via Hibernate provides the desired object-oriented view of the DBMS. Furthermore, the Person mapping file enables the prototype DB application to offer the object-oriented characteristic of inheritance. An abbreviated Person.hbm.xml file is provided below:

```
...
<hibernate-mapping>
        <class name="DBObjects.Person" table="per">
        <id name="id" column="per_id">
                        <generator class="native"/>
                </id>
                <property name="firstname"/>
                ...
                <property name="zip"/>

                <joined-subclass name="DBObjects.ServiceMember" table="sm">
                        <key column="per_id"/>
                        <property name="rank"/>
                        ...
                        <property name="deployed"/>
                        <set name="dependents" table="sm_dep">
                                <key column="sm_id"/>
                                <many-to-many column="dep_id" class =
                                "DBObjects.Dependent"/>
                        </set>
                        <set name="MOS" inverse="true" table="MOS_SM">
                                <key column="sm_id"/>
                                <many-to-many column="mos_id" class =
                                "DBObjects.MOS"/>
                        </set>
                </joined-subclass>
                <joined-subclass name="DBObjects.Dependent" table="dep">
                ...
                </joined-subclass>
        </class>

</hibernate-mapping>
```

The Person mapping file above begins by showing the direct correlation between the Person class and the corresponding table (“per”) in PostgreSQL. The mapping file then captures the Person class attributes (firstname, zip, etc.) that are also columns of the “per” table. The mapping file demonstrates inheritance between Person and ServiceMember (and between Person and Dependent) by joining their respective tables and noting the ServiceMember class as a subclass of Person. This demonstration of inheritance models the inheritance of the Person class and its subclasses in both Java and the RDBMS. Furthermore, this mapping file captures the relationships between the tables and the Java classes. Specifically, the many-to-many relationship is represented between the ServiceMember and the MOS objects and their corresponding tables in the Person mapping file.

DBObjects is a package of Java classes consisting of the Person, ServiceMember, Dependent, MOS, JCode, and TaskForce classes. Additionally, these classes are represented in both a PostgreSQL table and an XML mapping file. The relationships between the classes and the tables are consistent with the schema presented in the ER diagram provided (many-to-many, one-to-many, etc.). As already discussed, the table relationships are captured in the XML mapping files, however; the java class relationships are captured in the classes themselves. A portion of the ServiceMember class is provided to demonstrate the many-to-many relationship between objects:

```
public class ServiceMember extends Person implements java.io.Serializable {

        ...

        //basic constructor
        public ServiceMember(){
                super();
                dependents = new HashSet();
                MOS = new HashSet();
        }

        ...
        protected Set getMOS(){
                return MOS;
        }

        protected void setMOS(Set MOS){
                this.MOS=MOS;
        }

        public void addToMOS(MOS mos){
                this.getMOS().add(mos);
                mos.getServiceMembers().add(this);
        }

        ...
}
```

All DBObject classes in the prototype DB application have an appropriate getter and setter method for each attribute. For example, the ServiceMember object above has a set of MOS objects and an appropriate getter and setter method for that set of MOS objects. Further, the set of MOS objects is used to capture the many MOS objects that can relate to a given ServiceMember object. The addToMOS( ) method above adds the set of MOS objects to the ServiceMember object. Furthermore, the addToMOS( ) method then adds the current ServiceMember object to the set of MOS objects making the relationship between the ServiceMember and MOS objects bidirectional. Of note, the bidirectional relationship is also represented in the Person XML mapping file by showing at least one side of the relationship to contain the "inverse=true" statement. Other relationships are modeled in a similar fashion in both the java classes and the XML mapping files and will not be further discussed.

4. Presentation Layer

The presentation layer is where the user interfaces with the prototype DB application via a GUI. This layer, as seen in the Implemented Prototype DB Application Architecture diagram (Figure 15), consists of a PCRClient running on a PC or a PDA. The presentation layer receives user input via the GUI and converts that input to a Command object. The Command object is then sent via a HTTP connection to the PCRServer and receives a list of objects in return. This list of objects is then displayed on the GUI to the user. This section will provide a description of the PCRClient to include a UML diagram and critical segments of program code. The section will then conclude with a brief discussion of the PDA's specific implementation.

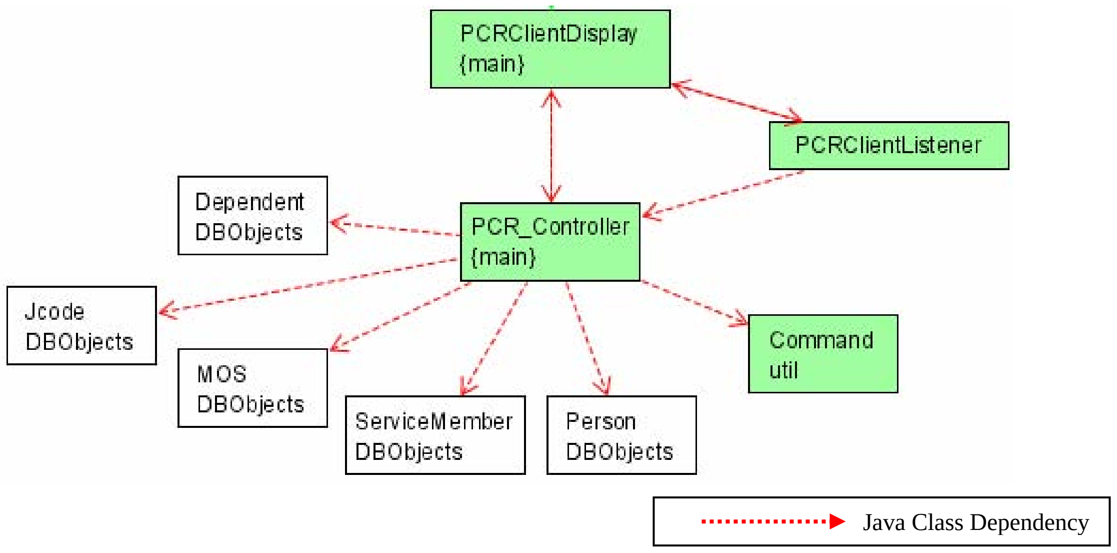
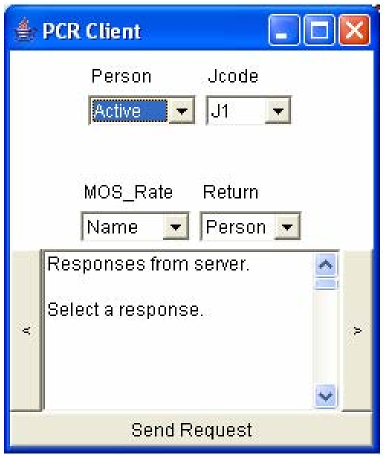


Figure 17. Presentation Layer UML Diagram

The PCRClientDisplay in the UML diagram (Figure 17) above provides the GUI functionality of the PCRClient. PCRClientDisplay awaits input from the user via the PCRClientListener then passes that input to the PCR_Controller where it is converted to a Command object. Additionally, the PCR_Controller contains all of the program logic and I/O of the PCRClient. The PCRClient will be described following the Presentation Layer Implementation diagram, Figure 14, to include the GUI, Client Logic, and I/O. Furthermore, the DBObjects above are identical to those found in the application logic layer.



Responses from server.

Select a response.

Figure 18. PCRClient GUI

The goal of the PCRClient GUI (Figure 18) is to provide an object-oriented view of the RDBMS and an intuitive means of accessing the data to the user. The GUI went through several iterations of design to collect user input ranging from a single touch button interface to the drop-down menu interface as seen above. Additionally, the small size of the PDA client display was a major design consideration in regards to both the input and output portions of the GUI. The GUI's dimensions are set to an appropriate size for both a PC and a PDA (iPAQ). This was accomplished by allowing the GUI to resize itself respective to the specific device's display that it is being executed on. The following segment of code from the PCRClientDisplay shows how this was accomplished:

```
/* Adjusts the size of the PCR Client interface in accordance with
* with device in use.  Ensures that the PCRClient frame is 90% of
* the width and 90% of the height. of the display. */
private void resize(){
     f.pack();

     //Set frame to the screen size
     Toolkit toolkit          = Toolkit.getDefaultToolkit();
     Dimension fullScreenSize = toolkit.getScreenSize();
     f.setSize(fullScreenSize.width - (int) (fullScreenSize.width *.1),
            fullScreenSize.height - (int)(fullScreenSize.height *.1));

     f.show();
}
```

The resize( ) method of the PCRClientDisplay above sets the GUI display to 90% of the client's display. This method begins by calling the pack( ) method and applying it to an Abstract Windowing Toolkit (AWT) Frame, specifically "f". The pack( ) method sizes the GUI window to a standard window size. The resize( ) method then resizes the GUI to 90% of the overall device's display. In addition to making the GUI properly fit the screen of the targeted device, the GUI implementation was limited to JDK 1.1 libraries as discussed previously. The mandated use of JDK 1.1 to accommodate a PDA client limited the GUI implementation to the Java AWT package vice the more current Java Swing package.

The PCRClient logic and I/O is implemented in PCR_Controller which has three main functions; (1) to creates the Command object, (2) to handle the I/O, and (3) to format the returning list for GUI display. The below code segment is where the Command object is created based on user input from the GUI:

```
public void beginSequence() {

    //checks conn
   establishConnection();

   //person
   String person = display.getPerson();

  //jcode
   String jcode = display.getJcode();

   //mos
   String mos = display.getMos();

    //return type
   String returnType = display.getReturnType();

    //create and fill selection filter
   String[] selectionFilter = new String[3];
   selectionFilter[0] = jcode; //("j1"-"j7") or "none"
   selectionFilter[1] = person; //("active" or "depend") or "none"
```

```
        selectionFilter[2] = mos; //("num" or "name") or "none"

        //create and pass cmd client
        Command clientCommand = new Command(1, returnType, selectionFilter);

        sendAndCheckResponse(clientCommand);
    }
```

The PCR_Controller's beginSequence( ) method above is initiated when the user selects the "send request" button on the GUI. The establishConnection( ) method is then used to create a connection to the Apache HttpServlet via an URL. The beginSequence method then retrieves the user's selections from the GUI drop-down menus which are then saved into the selectionFilter array or object returnType as appropriate. The selectionFilter captures the user's query request and the returnType specifies the type of objects returned from the Apache HttpServlet. The Command object is passed three variables in its creation; (1) the command type ("1" specifying a query, note: all Command objects are query objects for the scope of this thesis), (2) the object returnType, and (3) the selectionFilter. That Command object is then sent to the Apache HttpServlet via the sendAndCheckResponses( ) method as seen below:

```
    public void sendAndCheckResponse(Command cmd) {

             try {
                conn.setDoOutput(true);
                conn.setDoInput(true);

                   //Outgoing
                ObjectOutputStream out=new ObjectOutputStream(conn.getOutputStream());
                out.writeObject(cmd);

                out.flush();
                out.close();

                   //Incoming Stream
                ObjectInputStream in = new ObjectInputStream(conn.getInputStream() );

                   // Incoming List
                List inList = (List)in.readObject();
                in.close();

                   //Display type on gui
                String objRequested = cmd.getObjType();
                String[] request = cmd.getFilter();
                String status = request[1];
                 ...

                   //Pass List to correct output method
                if(objRequested.equals("Person") && status.equals("Active")){
                   smOutput(inList);
                }

                ... //additional if statements
                }catch (Exception e) {
                   ...
                }
          }
```

The PCR_Controller’s method above is the compliment to the PCRServer’s doPost( ) method. The sendAndCheckResponse( ) method begins by allowing for the URLConnection (“conn”) to accept input and output. The method then specifies the output of the URLConnection as an ObjectOutputStream and transmits the newly created Command object. After the ObjectOutputStream is flushed and closed, the input of the URLConnection is in turn specified as an ObjectInputStream. That ObjectInputStream then receives the list of objects from the PCRServer and is closed. Upon receiving the list of objects from PCRServer, the Command object that initiated the method call (“cmd”) is used to specify the type of objects and user’s display requirement. For example, if the type of objects requested are of type Person and they are on active duty then the list is passed to the smOutput( ) method for proper GUI display. Likewise, lists containing different object types are handled similarly via their respective display method(s).

The PDA’s JVM (Jeode) introduces specific requirements, namely the necessary Linker file. The Jeode JVM running on the iPAQ accepts inputs via this Linker file. Further, the Linker file allows for one-click access to PCRClient execution preventing the need for troublesome manual PDA input(s) via a stylus. This Linker file injects those inputs to the JVM for the user; the Linker file for this application is PCRClientDisplay.lnk shown below:

```
18#"\Windows\evm.exe" -Djeode.evm.console.local.keep=TRUE -cp \Windows\lib\Demo3
PCRClientDisplay
```

The Linker file above is used to keep the Java application open on the PDA by setting the console variable to true. Additionally this file directs Jeode to the location of the main class of the application, PCRClientDisplay. Further, this Linker file can specify Java Archive (JAR) files to a classpath. These JAR files allow for increased flexibility to the developer by introducing the potential for extended libraries.

E. SAMPLE INTERACTION

The following sample interaction illustrates the flow of commands and data among the different layers. The sample interaction will capture the request for Person objects with the following selection criteria: active duty service member, in the J7, with no MOS displayed. The sample interaction will follow the diagram below (Figure 19) in numeric sequence expounding upon critical points of the sample interaction beginning by the user using the GUI to request information and result in the requested information presented to the user via the GUI.

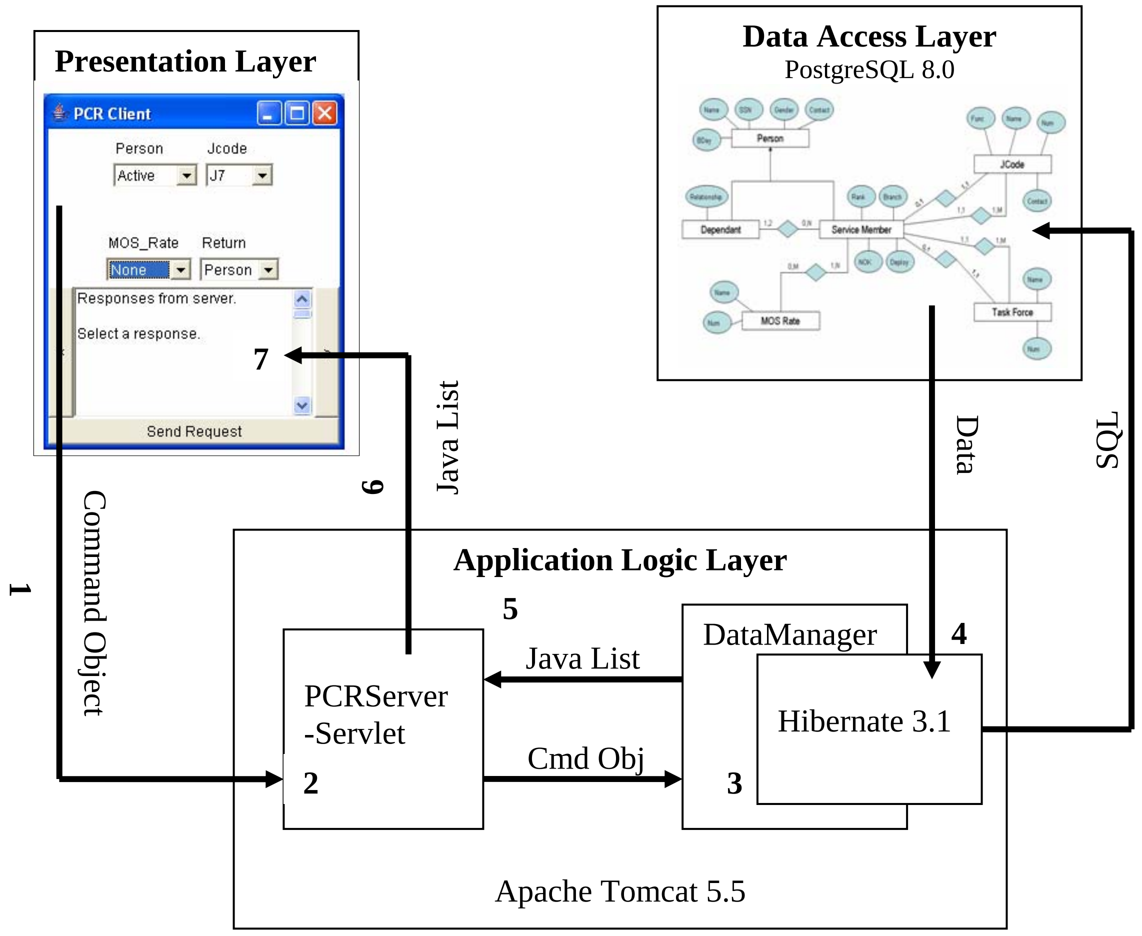


Figure 19. Prototype DB Application Sample Interaction

1. Command Object Creation and Transfer

The Command object is created in the PCR_Controller's beginSequence( ) method based upon the user's inputs (Active, J7, no MOS, Person return type, and query type Command object). The Command object is then transmitted to the PCRServer via the PCR_Controller's sendAndCheckResponse( ) method. This transmission occurs over an HTTP URL connection established between the PCR_Controller and the PCRServer.

2. PCRServer Receives the Command Object

PCRServer is a Java HttpServlet listening via the doPost( ) method over an HTTP URL connection. Once the doPost( ) method is invoked the Command object is received via an ObjectInputStream and then passed to an instance of DataManager's getResults( ) method. DataManager then processes the Command object using the OR Mapper.

3. DataManager and Hibernate Interaction

The DataManager receives the Command object via its getResults( ) method. Since this specific Command object is of type "query", it is passed to the getQuery( ) method where the user's request is converted to HQL. The HQL query generated by the getQuery( ) method is:

```
from Jcode j where j.number = 7.
```

That HQL query is then passed to the processQuery( ) method where a Hibernate Session is acquired from HibernateUtil. Lastly, that Session then creates a Hibernate Query object and receives the results via the Hibernate JDBC connection as a list of JCode objects (the J7 object) from PostgreSQL.

4. List of Objects Received from PostgreSQL

The List of JCode objects is returned to the processQuery( ) method of DataManager. Since the Command object specifies active duty ServiceMember objects as the return type, the processQuery( ) method will then capture the Set of ServiceMember objects in the J7. The J7 ServiceMember objects are then saved into a new java list and passed to the getCopied( ) method. The getCopied( ) method copies ServiceMember objects into a new list ensuring that only JDK 1.1 functionality remains. That new list is then returned to the getResults() method in DataManager where it was originally called from PCRServer.

5. PCRServer Receives List of Objects

Once the list of J7 ServiceMember objects is received by the PCRServer's instance of DataManager it is supplied to an ObjectOutputStream. The ObjectOutputStream in turn transmits the list from the PCRServer's doPost( ) method to the PCR_Controller. Finally, the ObjectOutputStream is closed upon transmission of the list of objects.

6. PCRClient Displays Results

The PCRController's sendAndCheckResponse( ) method receives the list of J7 ServiceMember objects over the existing HTTP URL connection established between the PCR_Controller and the PCRServer. Since the objects in the list are of type ServiceMember, the list is passed to the smOutput( ) method for proper display in the output window of the GUI. The specific data requested by the user (Active, J7, no MOS, and Person return type) is shown below as it is presented to the output portion of the GUI:

```
0 Rank: E-1 L_Name: servicemember9 SSN: 565258962 Sex: M
1 Rank: E-3 L_Name: servicemember14 SSN: 945971584 Sex: M
2 Rank: E-1 L_Name: servicemember18 SSN: 225123085 Sex: M
3 Rank: E-2 L_Name: servicemember11 SSN: 804804729 Sex: M
4 Rank: E-9 L_Name: servicemember8 SSN: 52940794 Sex: M
```

F. FINDINGS DURING IMPLEMENTATION

During the implementation of the proposed architecture there were several small problems encountered and overcome. There were two significant findings that had an impact on the DB application as a whole. The first significant finding was that the Set used by Hibernate was proprietary to Hibernate causing a conflict with the PCRClient. The second significant finding was encountered upon attempting to overcome that Hibernate proprietary Set. The remainder of this chapter will discuss both of these findings in more detail.

The first significant finding during implementation was that the Hibernate Set was not the same as the java.util.Set. This finding is significant for one primary reason. Hibernate utilizes the Set as a primary means of capturing the many side of a relationship; namely, the many-to-many, the many-to-one, or the one-to-many relationship. In the simplest of terms this is accomplished by each java object that contains a many relationship to another object capturing that many relationship via a Java Set. For example, the JCode table has a many-to-one relationship with the Service Member table as seen in the ER Diagram, Figure 12. Likewise each JCode object uses a Set of ServiceMember objects to represent that same many-to-one relationship. This at a glance seems to be no problem, however; when the PCRClient program ran and the result list reached the PCRClient an error was generated. Specifically:

```
        org.hibernate.collection.PersistentSet class java.lang.ClassNotFound.Exception
```

After troubleshooting the error was narrowed down to those very instances of the Java Set that were used by Hibernate to capture the many side of a relationship. This error was caused by a Hibernate version of the Set being used, namely the org.hibernate.collection.PersistentSet vice the expected java.util.Set.

Upon determing the cause of the error, two potential solutions to the problem were arrived at: (1) the objects contained in the Hibernate Set could be deep copied and saved into a java.util.Set in order to be recognized by the client or (2) the Hibernate3.jar file could be included in the classpath of the client in order to recognize the Hibernate version of the Java Set. The first solution, although viable, required additional code and could be potentially time consuming to incorporate into the prototype DB application. Thus, the clear choice was to include the Hibernate3.jar file in the client classpath.

Including the Hibernate3.jar file in the classpath was initially tested on a PC and went without a hitch producing the desire outputs to the PCRClient as expected. Including the Hibernate3.jar file on the PDA was much more complex, thus encountering the second significant finding during implementation. Initially, Hibernate3.jar was added to the PDA classpath via a Jeode Linker file as follows:

```
18#"\Windows\evm.exe" -Djeode.evm.console.local.keep=TRUE -cp
\Windows\lib\Demo3\hibernate3.jar;\Windows\lib\Demo3 PCRClientDisplay
```

The Linker file above failed to solve the problem as the Hibernate3.jar file is compiled to JDK 1.5 and the Jeode JVM is only capable of supporting JDK 1.1 or older. To overcome the inability to incorporate the Hibernate3.jar file into PCRClient on the PDA led to four possible solutions: (1) utilize a PDA JVM capable of supporting JDK 1.5, (2) select a new OR Mapper, (3) recompile the Hibernate3.jar file to JDK 1.1 or older, or (4) not use the Hibernate3.jar file on the PCRClient and deep copy the objects in the Hibernate Set to a java.util.Set as previously discussed.

Currently there are no PDA JVMs capable of supporting JDK 1.5, thus the first solution was dismissed. Further, the second solution of using a different OR Mapper was not acceptable due to the existing OR Mapper, Hibernate, being fully incorporated and functional with the PC based client. The third solution of compiling the Hibernate3.jar files to JDK 1.1 was attempted, but unsuccessful. Thus, the last solution of overcoming the Hibernate proprietary Set by deep copying was the only viable solution and implemented into the final prototype.

To deep copy the objects from the Hibernate proprietary Set to a java.util.Set not only required additional coding, but also added potential for performance limitations. The additional coding required was added to deep copy each object from the Hibernate Set. The deep copy operation entailed creating objects of the same type as those in the Hibernate Set, copying the contents of those objects into the new objects, and saving the new objects into a java.util.Set. The deep copy process removed the requirement for the Hibernate3.jar file to exist on the PCRClient device(s) and eliminated the org.hibernate.collection.PersistentSet error. Furthermore, deep copying was successful in overcoming the Hibernate Set problem, however; it increased the amount of data processing in the application logic layer. The increased processing stems from the need to deep copy each and every Hibernate Set that exists in the returning Hibernate query generated. Deep copying becomes very complex in cases where the Hibernate Set contains objects that in turn contain additional Hibernate Sets, and so on. The complexity compounded by the potential for large and complex query results potentially places a bottleneck on the application logic layer. Although there was a potential bottleneck in the application logic layer this was the clear solution in overcoming the Hibernate proprietary Set problem in the proof-of-concept prototype DB application.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SUMMARY

This chapter concludes the thesis with the general findings on the approaches used and the analysis of the work done in this thesis. The possible extensions to the thesis are also presented.

A. GENERAL FINDINGS AND ANALYSIS

The thesis's main objective was to overcome the conceptual gap between current software development technologies and the highly engrained Relational Database Management System (RDBMS) technologies. The additional objective was to support mobile device access to the DB application. In this section, general findings and analysis of the work done in this thesis will be presented.

1. OR Mapper

A method of overcoming the conceptual gap between the modern software development technologies and the RDBMS technologies was required. The chosen method of overcoming that gap was an Object Relational (OR) Mapper. The OR Mapper supplies a virtual Object Oriented Database Management System (OODBMS) that lies between a RDBMS and the rest of the DB application implemented with modern software development technologies. Specifically, the proof-of-concept prototype DB application utilized Hibernate as an OR Mapper during implementation because it was open source and properly supported.

During the implementation phase of this thesis, Hibernate provided sufficient support and capability for the following two reasons. First, though the learning curve to use Hibernate is initially steep it was quite powerful and adequate to meet the goals of an OR Mapper in a military application domain. The learning curve was overcome by leaning upon the online documentation and publications specific to Hibernate (e.g. Hibernate in Action). Second, Hibernate is capable of integrating into existing three-tier DB application architectures and not completely disrupting the pre-existing DB application software. The overall capabilities of Hibernate enable a modern Object-Oriented (OO) view of the RDBMS and thus overcome the conceptual gap.

2. Portability vs Bottleneck

Though Hibernate was fully capable of providing an OO view of the RDBMS it did not provide all the capabilities that its' documentation suggested. Specifically, the Hibernate documentation suggests that Plain Ordinary Java Objects (POJOs) are returned from the RDBMS. POJOs were returned in all cases encountered during implementation except one. The one case that did not return POJOs was the Java Set. As discussed in Chapter IV, Hibernate uses its own version of the Java Set, org.hibernate.collection.PersistentSet, which is clearly not a POJO. The use of this proprietary Java Set required the Hibernate3.jar file to be included on both the software client and server. The requirement that the Java Archive (JAR) file be included on the software client leads to portability and bottleneck considerations.

Mobile devices executing the client software necessitated the need to bypass the JAR file as discussed in Chapter IV. The method chosen to bypass the JAR file for mobile clients was deep copying the returning objects and converting them to true POJOs. The deep copy method will most likely degrade the rate of data throughput on the DB application due to the strong possibility for multiple, large, complex queries. This performance degredation was an unavoidable penalty incurred during implementation because it provided for the portability of the client software to mobile devices. However, to eliminate this problem in the prototype DB application required the use of the JAR file on the client system which was not practical for mobile devices.

3. Overall Critique of Work

The overall goal of the work presented in this thesis was to provide a means of overcoming the described conceptual gap and provide for mobile device access to DB applications in the military. Although the overall goal was achieved in the proof-of-concept prototype described in Chapter IV there were other alternative approaches to achieve the same goal. These alternative approaches will be further discussed in comparison to the implemented prototype DB application.

The prototype DB application incorporated a three-tier software architecture. While the three-tier software architecture met the requirements for the prototype, it may not be robust enough for all DB applications in a military environment. Specifically, an additional tier may have provided a more efficient means to incorporate mobile devices into the DB application. By using an additional tier (fourth tier) the potential performance degradation previously discussed can be minimized to a more acceptable level. The additional tier could determine if the software client is capable of utilizing a Hibernate JAR file, thereby eliminating the deep copy requirement for that client. In the case that the software client is not capable of utilizing the Hibernate JAR file (mobile device) the fourth tier performs the work of the deep copy. Thus, the addition of a fourth tier would improve the overall performance of the DB application when compared to the three-tier software architecture.

Additionally, the prototype DB application incorporated a full software application as a means of developing the software client. Though the full software application approach fully met the requirements of the prototype it presents a limitation to large scale DB application employment. A full software application is resident on each remote client requiring the client's software to be updated and maintained individually. One solution to bypassing this large scale software maintenance overhead is to utilize a technology such as a Java Applet. Although, the Java Applet does not provide the same degree of functionality (discussed in Chapter II) as the full software application it does not incur this software maintenance overhead.

B. FUTURE WORK

The presented proof-of-concept prototype DB application requires additional work in order to provide large scale use in today's military. The future work should extend the prototype DB application in three specific areas; (1) the Client Software (GUI), (2) map to an existing military RDBMS, and (3) a generalized prototype DB application. Each of these areas will be further discussed.

1. Client Software

The client software as presented in Chapter IV is somewhat limited in its functionality. To become more relevant to modern military applications the client software should become more robust in both functionality and usability. This increase in functionality and usability will require at a minimum three explicit improvements to the client software: (1) more precise queries, (2) a more dynamic GUI, and (3) additional functions of update, edit, and insertion.

First, the client software should provide the ability to define a more detailed and precise query. This precise querying ability will permit the user to capture the necessary information from the backend DBMS in a timely and efficient manner. Additionally, the user should be enabled to further search on the objects returned during the initial and any subsequent queries. Second, the client software must also provide the user with a more dynamic GUI for both input and output functions. These GUI improvements may be accomplished by adding additional features such as top level menus (i.e. File, Edit, Tools, View, etc.) and additional screens, or states (e.g. an output screen vice the existing output text field). These additional features will provide an overall increase in system usability. Lastly, the prototype proof-of-concept DB application is restricted to being read only. The client software functions must be expanded to allow for updates, edits, and insertions into the DBMS from the remote client providing more functionality to the user. Furthermore, each of these improvements to the client software must be implemented with mobile devices in mind.

2. Map to Existing Military RDBMS

The proof-of-concept DB application presented in Chapter IV maps to a small scale fictitious RDBMS that modeled a Joint Staff. Future DB applications of the proof-of-concept should be capable of mapping to a real world military RDBMS yielding a DB application that is both relevant and realistic. Mapping to an existing military RDBMS implicitly requires that the DB application take into account mobile device restrictions; namely, memory, processing power, screen size, etc. These mobile device restrictions in conjunction with the potential for large query results introduce the necessity for additional DB application logic; for example, the returning results may need to be filtered to an appropriate size for a mobile device. Thus, the proof-of-concept DB application requires additional logic and functionality in order to properly map to an existing military RDBMS.

3. Generalized Prototype DB Application

All portions of the proof-of-concept DB application are specific to only one RDBMS representing a Joint Staff. Thus, the overall DB application is not designed for general use. The DB application should be able to map to more than just one specific military RDBMS, it should be capable of mapping to all military RDBMSs with little modification to existing software. To achieve a more general DB application, there is a requirement that the application as a whole be more dynamic. The DB application must be generalized at two points; the presentation layer and the application logic layer.

The presentation layer should be more loosely coupled to the rest of the DB application. This requires that the presentation layer software query the rest of the DB application and configure the GUI appropriately. By dynamically configuring the GUI the presentation layer is fully capable of presenting the user a graphical representation of any backend RDBMS. By providing the DB application a loosely coupled presentation layer the developer is not required to reconfigure the DB application according to a specific backend RDBMS(s).

In addition to the presentation layer, the application logic layer must also be more generalized. A more generalized application logic layer requires the capability to first map to any RDBMS. In turn this requires the OR Mapper to generate objects according to the schema that is resident in the RDBMS. For example, the OR Mapper should be capable of connecting to a RDBMS and return the appropriate object structure that relates to the schema resident in that RDBMS.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Ambler, S. W. (2006). *Relational databases 101: Looking at the whole picture.* Retrieved July 7, 2006 from http://www.agiledata.org/essays/relationalDatabases.html#BeyondRelationalDatabases

Bertino, E., & Martino, L. (1993). *Object-oriented database systems concepts and architectures*. Wokingham, England: Addison-Wesley Publishers Ltd.

Elmasri, R., & Navathe, S. B. (2004). In Suarez-Rivas M., Harutunian K. (Eds.), *Fundamentals of database systems* (Fourth ed.). Boston, MA: Pearson Education, Inc.

Leavitt Communications, I. (2000). *Whatever happened to obect-oriented databases?* Retrieved July 5, 2006 from http://www.leavcom.com/db_08_00.htm

Microsoft TechNet. (2005). *How to configure web browser compatibility properties in FrontPage 2002.* Retrieved July 12, 2006 from http://support.microsoft.com/default.aspx?scid=kb;en-us;311341&sd=tech

Ramakrishnan, R., & Gehrke, J. (2003). In Lupash E. (Ed.), *Database management systems* (Third ed.). New York, NY: McGraw-Hill Companies, Inc.

Scott, A. W. (2006). *The object-relational impedance mismatch.* Retrieved July 10, 2006 from http://www.agiledata.org/essays/impedanceMismatch.html

StoneBraker, M., & Moore, D. (1996). In Spatz B. M., Morgan M. B. (Eds.), *Object-relational DBMS the next great wave*. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

THIS PAGE INTENTIONALLY LEFT BLANK

BIBLIOGRAPHY

Apache Tomcat Corporation. (2005). *The apache tomcat 5.5 Servlet/JSP container.*, 2005 from http://tomcat.apache.org/tomcat-5.5-doc/index.html

Bauer, C., & King, G. (2005). *Hibernate in action*. Greenwich, CT: Manning Publications Co.

Bertino, E., & Martino, L. (1993). *Object-oriented database systems concepts and architectures*. Wokingham, England: Addison-Wesley Publishers Ltd.

Cattell, R. G. G., Barry, D., Bartels, D., Berler, M., Eastman, J., & Gamerman, S., et al. (1997). *The object database standard ODMG 2.0*. San Fransisco, CA: Morgan Kaufmann Publishers, Inc.

Devarakonda, R. S. *Object-relational DBMSs - the road ahead.* Retrieved July 15, 2006 from http://www.acm.org/crossroads/xrds7-3/ordbms.html

Gray, P., Kulkarni, K., & Paton, N. (1992). *Object-oriented databases A semantic data model approach*. New York, New York: Prentice Hall.

Hibernate. *Hibernate - relational persistance for idiomatic java.*, 2005 from http://www.hibernate.org/hib_docs/v3/reference/en/html/

Lant, M. (2002). *Where to put the business rules.* Retrieved July 11, 2006 from http://www.sphere-data.com/docs/bus_rule.shtml

Larman, C. (2005). *Applying UML and patterns an introduction to object-oriented analysis and design and iterative development* (Third ed.). Upper Saddle River, NJ: Prentice Hall PTR.

Lausen, G., & Vossen, G. (1993). *Object-oriented databases concepts and architectures*. Reading, MA: Addison-Wesley Publishing Company Inc.

Obasanjo, D. (2001). *An exploration of object oriented database management systems.* Retrieved July 21, 2006 from http://www.25hoursaday.com/WhyArentYouUsingAnOODBMS.html

PostgreSQL. (2006). *PostgreSQL.*, 2005 from http://www.postgresql.org/

Snyder, M., & O'Connor, T. (2005). Object-relational mapping in java with SimpleORM. *Dr. Dobb's Journal, 379* (December 2005), 34-35-36.

Weiss, S. (2002). *Handheld usability*. New York: John Wiley & Sons, LTD.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California