



2000-03

Design, implementation, and analysis of the
Personnel, Operations, Equipment, and Training
(POET) database and application program for the
Turkish Navy Frigate

Can, Yuksel

Monterey California Naval Postgraduate School



Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

**Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943**

**NAVAL POSTGRADUATE SCHOOL
Monterey, California**



THESIS

**DESIGN, IMPLEMENTATION, AND ANALYSIS OF THE
PERSONNEL, OPERATIONS, EQUIPMENT, AND TRAINING (POET)
DATABASE AND APPLICATION PROGRAM FOR THE
TURKISH NAVY FRIGATES**

by

Yuksel Can

March 2000

Thesis Advisors:

Thomas Wu
Lee Edwards

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 4

20000622 028

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE: DESIGN, IMPLEMENTATION, AND ANALYSIS OF THE PERSONNEL, OPERATIONS, EQUIPMENT, AND TRAINING (POET) DATABASE AND APPLICATION PROGRAM FOR THE TURKISH NAVY FRIGATES			5. FUNDING NUMBERS
6. AUTHOR Can, Yuksel			
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)			10. SPONSORING / MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the authors 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) The Turkish Navy frigates have a challenging mission, which encompasses tactical, operational and administrative tasks. Lacking an automated information infrastructure hinders the ships' ability to efficiently perform the administrative activities, to generate the required reports quickly and to make effective decisions based on this information. The objective of this thesis is to design and implement the Personnel, Operations, Equipment, and Training (POET) Database and Application Program for the Turkish Navy frigates and to analyze the potential benefits that will be obtained by using this system. The POET database system will provide the Turkish Navy frigates with an automated information system that will support the administrative activities, release manpower to perform other duties and reduce the productive power loss by increasing the availability, accuracy, and consistency of the data. The thesis covers the analysis of requirements, conceptual database design using Semantic Data Model, logical database design on Microsoft Access DBMS, and implementation of the application program using Java and JDBC API. The result of this study is a functional application that will eliminate most of the current problems onboard the frigates and result in considerable savings of personnel power and time while providing the required information to the command quickly.			
14. SUBJECT TERMS Database, Relational Database System, Semantic Data Model, Java, JDBC, System Maintenance, Design, Implementation and Analysis of Information Systems			15. NUMBER OF PAGES 293
			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. Z39-18 298-102

Approved for public release; distribution is unlimited

**DESIGN, IMPLEMENTATION, AND ANALYSIS OF THE
PERSONNEL, OPERATIONS, EQUIPMENT, AND TRAINING (POET)
DATABASE AND APPLICATION PROGRAM FOR THE
TURKISH NAVY FRIGATES**

Yuksel Can
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1994


Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
and
MASTER OF SCIENCE IN SYSTEMS MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author:


Yuksel Can

Approved by:


C. Thomas Wu, Thesis Advisor


Lee Edwards, Thesis Advisor


Dan Boger, Chairman, Department of Computer Science


Reuben Harris, Chairman, Systems Management Department

ABSTRACT

The Turkish Navy frigates have a challenging mission, which encompasses tactical, operational and administrative tasks. Lacking an automated information infrastructure hinders the ships' ability to efficiently perform the administrative activities, to generate the required reports quickly and to make effective decisions based on this information.

The objective of this thesis is to design and implement the Personnel, Operations, Equipment, and Training (POET) Database and Application Program for the Turkish Navy frigates and to analyze the potential benefits that will be obtained by using this system. The POET database system will provide the Turkish Navy frigates with an automated information system that will support the administrative activities, release manpower to perform other duties and reduce the productive power loss by increasing the availability, accuracy, and consistency of the data.

The thesis covers the analysis of requirements, conceptual database design using Semantic Data Model, logical database design on Microsoft Access DBMS, and implementation of the application program using Java and JDBC API. The result of this study is a functional application that will eliminate most of the current problems onboard the frigates and result in considerable savings of personnel power and time while providing the required information to the command quickly.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
A. BACKGROUND	1
B. OBJECTIVE	2
C. METHODOLOGY	3
1. Requirements Analysis	3
2. Conceptual Database Design	4
3. Logical Database Design	4
4. Physical Database Design	5
5. System Analysis and Evaluation.....	5
D. ORGANIZATION OF THESIS	5
II. BACKGROUND.....	9
A. DATABASE SYSTEMS	9
1. Benefits of the Database Approach.....	11
2. Data Models, Schemas, and Instances	20
3. DBMS Architecture	22
B. THE RELATIONAL DATABASE MODEL.....	23
1. Relational Model Concepts.....	24
2. Relational Model Constraints	31
3. Update Operations on Relations	34
4. Relational Algebra	36
C. STRUCTURED QUERY LANGUAGE	40
1. Data Definition in SQL.....	41
2. Queries in SQL	43
3. Update Statements in SQL.....	46
4. Views in SQL.....	48

5.	Processing SQL Statements	49
6.	SQL Techniques.....	51
D.	NORMALIZATION	56
1.	Functional Dependencies	57
2.	Keys	59
3.	Update Anomalies.....	60
4.	Normal Forms	62
5.	Summary	69
E.	ACCESS 97	70
1.	Features of Access 97	71
2.	Requirements for Access 97	77
3.	Database Objects and Views in Access 97	78
III.	SEMANTIC DATA MODEL.....	83
A.	INTRODUCTION	83
B.	SEMANTIC OBJECTS	85
1.	Attributes.....	86
2.	Attribute Cardinality	88
3.	Paired Attributes	89
4.	Object Identifiers	89
5.	Attribute Domains.....	90
C.	TYPES OF SEMANTIC OBJECTS.....	90
1.	Simple Objects	90
2.	Composite Objects	91
3.	Compound Objects.....	92
4.	Hybrid Objects	92
5.	Association Objects	93

6.	Parent/Subtype Objects.....	95
7.	Archetype/Version Objects.....	97
D.	TRANSFORMATION OF SEMANTIC OBJECTS INTO RELATIONS	98
1.	Transformation of Simple Objects.....	98
2.	Transformation of Composite Objects.....	99
3.	Transformation of Compound Objects	100
4.	Transformation of Hybrid Objects.....	103
5.	Transformation of Association Objects	106
6.	Transformation of Parent/Subtype Objects.....	108
7.	Transformation of Archetype/Version Objects.....	109
IV.	JAVA AND JAVA DATABASE CONNECTIVITY (JDBC).....	111
A.	JAVA	111
B.	ADVANTAGES OF JAVA.....	113
1.	Java is Portable	114
2.	Java is Object-Oriented.....	114
3.	Java Makes It Easy to Write Correct Code.....	115
4.	Java Includes a Library of Classes and Interfaces	117
5.	Java is Extensible.....	118
6.	Java is Secure.....	118
7.	Java is Multithreaded	119
8.	Java Performs Well.....	120
9.	Java Scales Well	120
10.	Java is Distributed.....	121
11.	Java is Robust	121
12.	Java is Dynamic	121

C.	JDBC.....	122
1.	Loading the Driver.....	122
2.	Establishing a Connection with the Database.....	123
3.	Sending SQL Statements	123
4.	Processing the Results.....	124
D.	JDBC CLASSES AND INTERFACES.....	125
1.	DriverManager Class	125
2.	Connection Interface.....	126
3.	Statement Interface	126
4.	PreparedStatement Interface	128
5.	CallableStatement Interface.....	129
6.	ResultSet Interface.....	129
7.	ResultSetMetaData Interface	131
8.	DatabaseMetaData Interface.....	132
E.	JDBC AND CLIENT/SERVER MODELS.....	133
F.	JDBC DRIVERS.....	135
1.	JDBC-ODBC Bridge plus ODBC Driver	136
2.	Native-API partly-Java Driver.....	137
3.	JDBC-Net pure Java Driver	138
4.	Native-protocol pure Java Driver.....	138
5.	Driver Selection	139
V.	REQUIREMENTS ANALYSIS FOR POET DATABASE	141
A.	DATABASE DEVELOPMENT PROCESS	141
1.	Requirements Collection and Analysis.....	141
2.	Conceptual Database Design	144
3.	Logical Database Design	145

4.	Physical Database Design.....	146
B.	REQUIREMENTS ANALYSIS FOR POET DATABASE.....	147
1.	Ship Object.....	147
2.	Department Object.....	148
3.	Division Object.....	148
4.	Personnel Object.....	148
5.	Training Object.....	150
6.	Operation Object.....	150
7.	Equipment Object.....	151
C.	DATA DICTIONARY FOR POET DATABASE.....	152
VI.	LOGICAL DATABASE DESIGN FOR POET DATABASE.....	153
A.	RELATIONAL TABLES OF POET DATABASE.....	153
1.	Ship Relation.....	155
2.	Overhauls Relation.....	156
3.	Department Relation.....	156
4.	Division Relation.....	157
5.	Personnel Relation.....	157
6.	Courses-To-Take Relation.....	158
7.	Courses-Taken Relation.....	159
8.	Assignments Relation.....	159
9.	Foreign-Languages Relation.....	160
10.	Training Relation.....	160
11.	Operation Relation.....	161
12.	Events Relation.....	161
13.	Port-Visits Relation.....	162
14.	Equipment Relation.....	162
15.	Failures Relation.....	163

B.	POET DATABASE RELATIONSHIPS	163
VII.	IMPLEMENTATION OF POET DATABASE AND DEVELOPMENT OF APPLICATION PROGRAM.....	165
A.	POET DATABASE IMPLEMENTATION	165
B.	APPLICATION PROGRAM IMPLEMENTATION.....	168
1.	Input Forms.....	170
2.	Update Forms.....	171
3.	Tables.....	173
4.	Reports.....	174
5.	Queries.....	175
VIII.	SYSTEMS IMPLEMENTATION AND SUPPORT.....	179
A.	SYSTEMS MAINTENANCE	180
B.	QUALITY ASSURANCE.....	182
1.	Testing.....	182
2.	Verification	182
3.	Validation.....	183
4.	Certification	183
5.	Testing Strategies.....	184
C.	TRAINING	185
D.	CONVERSION.....	187
1.	Parallel Systems	187
2.	Direct Conversion	188
3.	Pilot Approach	189

4.	Phase-In Method	189
E.	SYSTEMS RELIABILITY	190
IX.	ANALYSIS OF POET DATABASE SYSTEM	191
A.	CURRENT SITUATION.....	191
B.	FILE PROCESSING SYSTEMS.....	193
1.	Data Redundancy	193
2.	Data Inconsistency	193
3.	Limited Sharing of Data.....	194
4.	Program/Data Dependency	194
5.	Inflexibility of Information	195
6.	Data Isolation	195
7.	Difficulty in Representing Data.....	195
8.	Difficulty in Information Resource Management.....	196
C.	DATABASE PROCESSING SYSTEMS.....	196
1.	Minimum Data Redundancy	197
2.	Improved Data Sharing	198
3.	Increased Data Availability.....	198
4.	Cost Reduction.....	198
5.	Flexibility in Data Access.....	198
6.	Advanced Security and Integrity.....	199
7.	Program/Data Independence	199
8.	Dynamic Structure	200
D.	BENEFITS OF THE POET DATABASE SYSTEM.....	200
1.	Technical Aspect.....	200
2.	Manpower Aspect	201
3.	Decision Making Aspect.....	202

E.	INSTALLATION OF THE POET DATABASE SYSTEM	203
1.	Training.....	203
2.	Conversion.....	204
3.	Integration.....	206
F.	ASSESSING THE IMPACTS OF COMPUTER TECHNOLOGY IN ORGANIZATIONS	206
G.	CONCLUSION.....	208
X.	CONCLUSIONS	211
A.	SYNOPSIS.....	211
B.	FUTURE ENHANCEMENTS	213
	APPENDIX A: SEMANTIC OBJECTS.....	215
	APPENDIX B: DOMAIN SPECIFICATIONS.....	225
	APPENDIX C: RELATIONAL TABLES.....	239
	APPENDIX D: RELATIONSHIP DIAGRAM.....	243
	APPENDIX E: APPLICATION PROGRAM SCREEN SHOTS	245
	APPENDIX F: APPLICATION PROGRAM CODE	267
	LIST OF REFERENCES.....	429
	INITIAL DISTRIBUTION LIST	431

LIST OF FIGURES

Figure 2.1: University Database	26
Figure 2.2: STUDENT Relation	28
Figure 2.3: ACTIVITY Relation.....	60
Figure 3.1: Semantic Object Diagram	87
Figure 3.2: EQUIPMENT Simple Object.....	91
Figure 3.3: HOTEL-BILL Composite Object.....	91
Figure 3.4: BOOK and AUTHOR Compound Objects	92
Figure 3.5: SALES-ORDER Hybrid Object.....	93
Figure 3.6: FLIGHT, AIRPLANE, and PILOT Semantic Objects.....	94
Figure 3.7: EMPLOYEE Supertype and MANAGER Subtype Objects	95
Figure 3.8: Exclusive Subtypes.....	96
Figure 3.9: TEXTBOOK Archetype and EDITION Version Objects.....	97
Figure 3.10 (a): EQUIPMENT Simple Object	98
Figure 3.10 (b): EQUIPMENT Relation.....	98
Figure 3.11 (a): HOTEL-BILL Composite Object	99
Figure 3.11 (b): HOTEL-BILL and LINEITEM Relations	100
Figure 3.12 (a): One-to-One Compound Objects	101
Figure 3.12 (b): MEMBER and LOCKER Relations	101
Figure 3.13 (a): One-to-Many Compound Objects.....	102
Figure 3.13 (b): EQUIPMENT and REPAIR Relations	102
Figure 3.14 (a): BOOK and AUTHOR Compound Objects.....	103
Figure 3.14 (b): BOOK, AUTHOR, and BOOK-AUTHOR-INTERSECTION Relations	103
Figure 3.15 (a): SALES-ORDER Hybrid Object and ITEM, CUSTOMER and SALESPERSON Compound Objects	104
Figure 3.15 (b): SALES-ORDER, ITEM, CUSTOMER, SALESPERSON, and LINEITEM Relations	105

Figure 3.16 (a): FLIGHT Association Object and AIRPLANE and PILOT Compound Objects	107
Figure 3.16 (b): AIRPLANE, PILOT, and FLIGHT Relations	108
Figure 3.17 (a): EMPLOYEE Supertype and MANAGER Subtype Objects.....	108
Figure 3.17 (b): EMPLOYEE and MANAGER Relations.....	109
Figure 3.18 (a): TEXTBOOK Archetype and EDITION Version Objects	109
Figure 3.18 (b): TEXTBOOK and EDITION Relations.....	110
Figure 4.1: Typical JAVA Environment.....	113
Figure 4.2: JDBC Two-Tier Model	134
Figure 4.3: JDBC Three-Tier Model	135
Figure 4.4: JDBC Driver Implementation	136
Figure 6.1: Semantic Object – Relational Table Transformation.....	155
Figure 6.2: POET Database Relationship Diagram	164
Figure 7.1: Data Types Available in Microsoft Access.....	166
Figure 7.2: Table Design View for Operation Relation.....	167
Figure 7.3: QBE Window for Previous Assignments Query.....	168
Figure 7.4: POET Application Program Architecture	169
Figure 7.5: Operation Input Form.....	171
Figure 7.6: Select Exercise Dialog Box for Operation Update Form.....	172
Figure 7.7: Operation Update Form.....	172
Figure 7.8: Training Table	173
Figure 7.9: Port Visit Report.....	175
Figure 7.10: Select Exercise Dialog Box.....	177
Figure 7.11: Exercise/Event Query.....	177

LIST OF TABLES

Table 4.1: SQL and Java Data Types and Recommended Conversion Methods	130
---	-----

ACKNOWLEDGEMENT / DEDICATIONS

One of the great pleasures of finishing up this thesis is acknowledging the support of people whose names may not appear anywhere in the thesis, but whose cooperation, friendship, understanding and patience were crucial for me to prepare this thesis and successfully publish it.

I would like to extend my sincere gratitude to my thesis advisors, Professor C. Thomas Wu and Prof. Lee Edwards, for assisting me in deciding a thesis topic that will satisfy the requirements of both Computer Science and Systems Management departments, helping me throughout the study and making it a beneficial experience. Additionally, I would like to thank my wife, Sibel Can, for enduring the entire thesis process.

Finally, I would like to dedicate this thesis to my daughter, Rana Deniz Can, who is born during my thesis study.

I. INTRODUCTION

A. BACKGROUND

The Turkish Navy frigates, in their present state, lack the automated information infrastructure required to efficiently perform their administrative tasks. This hinders the ships' ability to generate the required reports rapidly and to make decisions effectively based on this administrative information. The lack of an adequate information technology system results in redundant and imprecise data maintained at different field sites and in different file formats. This ultimately leads to a waste of computer resources, manpower and time.

The management of the administrative activities is a difficult and time-consuming job in terms of report and message preparation, maintenance of data at different sites, and access to information. Furthermore, the large volume of daily, weekly, monthly, and annual reports required either for submission to the higher command or for the ship's internal use, makes the administrative tasks very difficult. In addition, the command needs timely and accurate information in decision making.

In the current situation, it is a time consuming process to prepare the required documents, because each department in the ship keeps its data in a different format and environment and the information needed is not stored in a central database. There is neither a standard format nor a software program to store, manipulate, and access the data.

As a solution to the problems discussed in the previous paragraphs, a database that will store information about Personnel, Operations, Equipment, and Training (POET) and an application program that will provide the graphical user interface will be developed for the Turkish Navy frigates. The POET database system will provide the Turkish Navy ships with an automated information system to perform their primary administrative functions. POET will support this mission by keeping track of all the personnel, operations, equipment, and training records, maintaining them, producing standard reports and providing the command with ad hoc information. This program is expected to eliminate most of the current problems and to result in considerable savings of personnel power and time while providing the required information to the command quickly.

B. OBJECTIVE

The objective of this thesis is to design and implement the Personnel, Operations, Equipment, and Training (POET) Database and Application Program for the Turkish Navy frigates and to analyze the potential benefits that will be obtained by using this system. The main goal of developing the POET database system is to support the administrative activities, to release manpower to perform other duties and to reduce the productive power loss by increasing the availability, accuracy, efficiency, and consistency of the data needed to generate the documents and reports. The use of the POET database system will greatly reduce the work hours spent on specific administrative tasks and provide more time to maintain an efficient operational level.

The design of the database system takes the Turkish Navy frigates' functional requirements into consideration. The primary function of the database system is to store the personnel, operations, equipment, training and other relevant information in a central database, to provide an easy-to-use graphical interface, to generate some standard reports and ad hoc queries, and to help the administrative office personnel.

C. METHODOLOGY

There are different methodologies for developing systems. The process that will be followed in this thesis captures the essence of most development methodologies. The fundamental phases of the system development process are explained briefly in the following subsections: Requirements Analysis; Conceptual Database Design; Logical Database Design; Physical Database Design; Systems Analysis and Evaluation.

1. Requirements Analysis

The major task of the first step in database development process is collecting information content and the processing requirements from all the identified and potential users of the database. During this step, database users are interviewed to understand and document the data requirements. In parallel with specifying the data requirements, it is useful to specify the known functional requirements of the application. During the requirements analysis phase, the tasks are to create the user's data model, determine the functional components of the application, and use prototypes to help determine user requirements.

2. Conceptual Database Design

Once all the requirements have been collected and analyzed, the next step is to create a conceptual schema for the database, using a high-level conceptual data model, such as Entity-Relationship Model or Semantic Data Model. The conceptual schema is a concise description of the data requirements of the user and includes detailed descriptions of the data types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Semantic Data Model will be used as the high-level data model to represent the conceptual schema for the POET database.

3. Logical Database Design

The next step in the database design is the actual implementation of the database, using a commercial DBMS. The major goal of the logical database design phase is to use the results of the conceptual design phase and the processing requirements as input to create a DBMS-processible schema as output. During this phase, the tasks are to develop the database design and the application design. The database design consists of structuring the relations, and establishing the relationships among them. The application design deals with the design of the forms, reports, and tables as well as the specification of update, display, and control mechanisms.

POET database system will be developed by using Microsoft Access database management system.

4. Physical Database Design

During the physical database design phase, the internal storage structures and file organizations for the database are specified. Physical database design is the process of developing an efficient and implementable physical database structure from a given logical database structure that has been shown to satisfy user information requirements. In parallel with these activities, application programs are implemented as database transactions corresponding to the high-level transaction specifications.

The application program will be implemented with Java programming language and JDBC application programming interface.

5. System Analysis and Evaluation

Upon completion of the implementation, POET database system will be evaluated and the possible benefits and advantages that would be gained by using the system will be analyzed from manpower, management, and technical perspectives. In this phase, systems implementation and support issues, such as conversion, training, testing, and systems reliability and maintenance, are discussed.

D. ORGANIZATION OF THESIS

This thesis is organized into the following chapters:

- Chapter I: Introduction. This chapter gives an overview of the problem, motivation, purpose and general outline of the thesis. It provides information about the background, objective, and methodology of the study.

- Chapter II: Background. This chapter is intended to provide an overview of the concepts used throughout the thesis. An explanation of the Database Systems, Relational Database Model, Structured Query Language, Normalization, and Microsoft Access Database Management System will be provided.
- Chapter III: Semantic Data Model. This chapter describes the Semantic Data Model, a high-level semantics-based data model that enables the semantics of a database to be incorporated directly into its schema. The semantic object types as well as the transformation of semantic objects into the relational tables are explained in this chapter.
- Chapter IV: Java and JDBC. Java and the JDBC package provide a concise and efficient way to access and manipulate data stored in a Relational Database Management System (RDBMS). The interaction between the user interface and back-end data sources of the POET database system is based on JDBC. This chapter will describe how to use Java and JDBC application programming interface (API) to provide this type of interaction. It will summarize the attributes of Java programming language and outline the JDBC API, classes, methods, and how they can be used by applications to directly access a RDBMS.

- Chapter V: Requirements Analysis for POET Database. This chapter first provides a general description of the database development process and briefly explains the phases of the process. Then, data requirements for the POET database system are explained by giving information about the semantic objects that constitute the data model.
- Chapter VI: Logical Database Design for POET Database. In this chapter, the logical database design for POET database is described. Logical database design phase covers the transformation of the semantic objects into the relational model. The POET database tables and the relationships among them are defined in Chapter VI.
- Chapter VII: Implementation for POET Database and Application Program. This chapter takes the reader through the database and application program design for the POET system. It explains how the relational database tables are implemented in Microsoft Access RDBMS and provides information about the forms, reports, tables, and queries supported by the application program.
- Chapter VIII: Systems Implementation and Support. This chapter will discuss the systems implementation and support issues in general. Five aspects of systems implementation and support; including system maintenance, quality assurance, system reliability, training, and conversion will be described.

- Chapter IX: Analysis of POET Database System. Chapter IX provides an analysis and evaluation of the POET database system. First, a brief introduction about the current situation of information processing in the Turkish Navy frigates is given and the file-processing systems is compared with the database processing systems. Then, it analyzes the benefits of the system from managerial, manpower, and technical aspects. Finally, the system implementation and installation issues are explained for the POET database system.
- Chapter X: Conclusions. This chapter provides a short summary of the thesis and addresses possible future enhancements that might be made to the developed system.
- Appendices A through F supplement the chapters by providing complete diagrams, specifications, and program code.

Appendix A: Semantic Objects

Appendix B: Domain Specifications

Appendix C: Relational Tables

Appendix D: Relationship Diagram

Appendix E: Application Program Screen Shots

Appendix F: Application Program Code

II. BACKGROUND

This chapter provides the background information necessary to understand the thesis and the POET database application program. Hence, information will be presented about database systems, relational database model, structured query language, normalization, Microsoft Access, and systems implementation and support in the following subsections.

A. DATABASE SYSTEMS

Databases and database technology are having a major impact on the growing use of computers. Databases play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and intelligence, to name a few. The word *database* is in such common use that we must begin by defining what a database is. A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit meaning.

The preceding definition of database is quite general; for example, one may consider the collection of words that make up this page of text to be related data and hence to constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties: [Ref. 1]

- A database represents some aspect of the real world, sometimes called the miniworld or the Universe of Discourse (UoD). Changes to the miniworld are

reflected in the database.

- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has a source from which data are derived, some degree of interaction with events in the real world, and an audience that is actively interested in the contents of the database. [Ref. 1]

A database management system (DBMS) is a collection of programs that enables users to create and maintain a database. The DBMS is a general-purpose software system that facilitates the processes of defining, constructing, and manipulating databases for various applications. *Defining* a database involves specifying the data types, structures, and constraints for the data to be stored in the database. *Constructing* the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. *Manipulating* a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data.

It is not necessary to use general-purpose DBMS software for implementing a computerized database. The programmer could write his or her own set of programs to

create and maintain the database, in effect creating a special-purpose DBMS software, as it is done in the implementation of the POET database application program. The database and the software together are called a *database system*.

1. Benefits of the Database Approach

A number of characteristics distinguish the database approach from the traditional approach of programming with files. In traditional file processing, each user defines and implements the files needed for a specific application. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain the data up-to-date.

In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users. The main properties of the database approach versus the file processing approach are described as follows.

a. Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself, but also a complete definition or description of the database. This definition is stored in the system catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called *metadata*, and it describes the structure of the primary database. [Ref. 1]

The catalog is used by the DBMS software and occasionally by database users, who need information about the database structure.

The DBMS software is not written for any specific database application, and hence it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access.

In traditional file processing, data definition is typically part of the application programs. Hence, these programs are constrained to work with only one specific database, whose structure is declared in the application programs. Whereas file-processing software can only access specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and then using these definitions.

b. Data Abstraction

In traditional file processing, the structure of data files is embedded in the access programs, so any changes to the structure of a file may require changing all programs that access this file. By contrast, DBMS access programs are written independently of any specific files. The structure of data files is stored in the DBMS catalog separately from the access programs. This property is normally called *program-data independence*.

Recent developments in object-oriented databases and programming languages allow users to define operations on data as part of the database definitions. An *operation* (also called a *function*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on

the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed *program-operation independence*. [Ref. 1]

The characteristic that allows program-data independence and program-operation independence is called *data abstraction*. A DBMS provides users with a conceptual representation of data that does not include many of the details of how the data is stored. Informally, a data model is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model hides storage details that are not of interest to most database users.

c. Support of Multiple Views of the Data

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files, but not explicitly stored. A multi-user DBMS whose users have a variety of applications provides facilities for defining multiple views. [Ref. 1]

d. Sharing of Data and Multi-user Transaction Processing

A multi-user DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include

concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct. An example is when several reservation clerks try to assign a seat on an airline flight; the DBMS should ensure that each seat can be accessed by only one clerk at a time for passenger assignment. These are generally called *transaction-processing applications*. A fundamental role of multi-user DBMS software is to ensure that concurrent transactions operate correctly without interference. [Ref. 1]

e. Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. Much of the data is stored twice: once in the files of each user group. Additional user groups may further duplicate some or all of the same data in their own files.

This redundancy in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update -- such as entering data on a new tuple -- multiple times. This leads to *duplication of effort*. Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent*. This may happen because an update is applied to some of the files, but not to others.

In the database approach, the views of different user groups are integrated during database design. For consistency, we should have a database design that stores each logical data item in only one place in the database. This does not permit any inconsistency and it saves storage space. [Ref. 1]

f. Restricting Unauthorized Access

When multiple users share a database, it is likely that some users will not be authorized to access all information in the database. For example, financial data is often considered confidential, and hence only authorized persons are allowed to access such data. In addition, some users may be permitted only to retrieve data, whereas others are allowed both to retrieve and to input; i.e., updates. Hence, the type of access operation -- retrieval or update -- must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a security and authorization subsystem, which the database administrator (DBA) uses to create accounts and to specify account restrictions. The DBMS should then enforce these restrictions automatically. [Ref. 1]

g. Persistent Storage for Program Objects and Data Structures

A recent application of databases is to provide persistent storage for program objects and data structures. This is one of the main reasons for the emergence of the object-oriented DBMS. Programming languages typically have complex data structures, such as record types in PASCAL or class definitions in C++. The values of program variables are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need again arises to read this data, the programmer must convert from the file format to the program variable structure.

Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be persistent, since it survives the termination of program execution and can later be directly retrieved by another C++ program. [Ref. 1]

h. Database Inferencing Using Deduction Rules

Another recent application of database systems is to provide capabilities for defining deduction rules for inferencing new information from the stored database facts. Such systems are called deductive database systems. For example, in education there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified declaratively as deduction rules, which when executed can determine all students on probation. In a traditional DBMS, an explicit procedural program code would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. [Ref. 1]

i. Providing Multiple User Interfaces

Because many types of users, with varying levels of technical knowledge, use a database, a DBMS should provide a variety of user interfaces. These include query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven

interfaces and natural language interfaces for stand-alone users. [Ref. 1]

j. Representing Complex Relationships Among Data

A database may include numerous varieties of data that are interrelated in many ways. A DBMS must have the capability to represent a variety of complex relationships among the data as well as to retrieve and update related data easily and efficiently. [Ref. 1]

k. Enforcing Integrity Constraints

Most database applications have certain integrity constraints that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. A more complex type of constraint that occurs frequently involves specifying that a record in one file must be related to records in other files. Another type of constraint specifies uniqueness on data item values. These constraints are derived from the meaning or semantics of the data and of the miniworld it represents. It is the database designers' responsibility to identify integrity constraints during database design. [Ref. 1]

l. Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The backup and recovery subsystem of the DBMS is responsible for recovery.

For example, if the computer system fails in the middle of a complex update program, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the program started executing.

Alternatively, the recovery subsystem could ensure that the program is resumed from the point at which it was interrupted so that its full effect is recorded in the database. [Ref. 1]

m. Potential for Enforcing Standards

The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own files and software. [Ref. 1]

n. Reduced Application Development Time

A prime feature of the database approach is that developing a new application takes very little time. Designing and implementing a new database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a traditional file system. [Ref. 1]

o. Flexibility

It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs additional information not currently in the database. In response, we may need to add a new file to the database or to extend the data elements in an existing file. Database systems allow such changes to the structure of the database without affecting the stored data and the existing application programs. [Ref. 1]

p. Availability of Up-to-Date Information

A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS. [Ref. 1]

q. Economies of Scale

The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments. This reduces overall costs of operation and management.

2. Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction by hiding details of data storage that are not needed by most database users. A data model is the main tool for providing this abstraction. A *data model* is a set of concepts that can be used to describe the structure of a database. By *structure* of a database, it is meant data types, relationships, and constraints are used to configure/organize the data. Most data models also include a set of basic operations for specifying retrievals and updates on the database. It is gradually becoming common practice to include concepts in the data model to specify behavior; this refers to specifying a set of valid user-defined operations that are allowed on the database in addition to the basic operations provided by the data model.

a. Categories of Data Models

It is possible to categorize data models based on the types of concepts they provide to describe the database structure. High-level or conceptual data models provide concepts that are close to the way many users perceive data, whereas low-level or physical data models provide concepts that describe the details of how data is stored in the computer. Between these two extremes is a class of representational (or implementation) data models, which provide concepts that may be understood by end users but that are not too far removed from the way data is organized within the computer. Representational data models hide some details of data storage, but can be implemented on a computer system in a direct way.

High-level data models use concepts such as entities, attributes, and relationships. An entity represents a real-world object or concept, such as an employee or a project, which is stored in the database. An attribute represents some property of interest that further describes an entity, such as the employee's name or salary. A relationship among two or more entities represents an interaction among the entities; for example, a works-on relationship between an employee and a project.

Representational or implementation data models are the ones used most frequently in current commercial DBMSs, and they include the four most widely used data models: Relational, network, hierarchical, and object-oriented. They represent data by using record structures and hence are sometimes called record-based data models. We can regard object-oriented data models as a new family of higher-level implementation data models that are closer to conceptual data models.

Physical data models describe how data is stored in the computer by representing information such as record formats, record orderings, and access paths. An access path is a structure that makes the search for particular database records efficient.

[Ref. 1]

b. Schemas and Instances

In any data model it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the database schema (or the metadata). A database schema is specified during database design and is not expected to change frequently.

However, the actual data in a database may change frequently. The data in the database at a particular moment in time is called a *database state* (or set of occurrences or instances). The distinction between database schema and database state is very important. When we define a new database, we only specify its database schema to the DBMS. At this point, the corresponding database state is the "empty state" with no data. The DBMS stores the schema in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to.

3. DBMS Architecture

Described in this section is the architecture for database systems, called the three-schema architecture that is proposed to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels: internal; conceptual; and external schema. [Ref. 1]

a. Internal Schema

The internal level has an internal schema, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

b. Conceptual Schema

The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users.

The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.

c. External Schema

The external level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

B. THE RELATIONAL DATABASE MODEL

Within the realm of database engineering, there are four basic types of database models: Relational, Network, Hierarchical, and Object-Oriented database models. The relational model represents the database as a collection of tables, where each table can be stored as a separate file. The network model represents data as record types and also represents a limited type of one-to-many relationship, called a set type. The network has an associated record-at-a-time language that must be embedded in a host programming language. The hierarchical model represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model, although most hierarchical DBMSs have record-at-a-time languages. The object-oriented model defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a class, and

classes are organized into hierarchies. The operations of each class are specified in terms of predefined procedures, called *methods*.

Most of the commercial database management systems implement the relational database model, which is the most common model in use today. Therefore, the focus of this section and the implementation of the POET database system will be the relational database model.

The relational model was introduced by E.F. Codd in 1970 and it is based on a simple and uniform data structure, called the *relation*, and has a solid theoretical foundation. The relational model represents the database as a collection of relations. Informally, each relation resembles a table or, to some extent, a simple file. [Ref. 1] For example, the database of tables shown in Figure 2.1 is considered to be in the relational model.

1. Relational Model Concepts

When a relation is thought of as a table of values, each row in the table represents a collection of related data values. These values can be interpreted as facts describing a real-world entity or relationship. The table name and column names are used to help in interpreting the meaning of the values in each row of the table. An example is presented here for explanation. The first table of Figure 2.1 is called STUDENT, because each row represents facts about a particular student entity. The column names - StudentName, StudentNumber, Class, Major - specify how to interpret the data values in each row, based on the column each value is in. All values in a column are of the same data type.

STUDENT Relation

StudentName	StudentNumber	Class	Major
Flowers	17	4	CS
Dowler	25	3	CS
Tidwell	36	4	EE

COURSE Relation

CourseName	CourseNumber	CreditHours	Department
Database	CS3320	4	CS
Networks	IS3502	4	ITM
Computer Security	CS3600	3	CS
Calculus	MA3200	5	MATH

SECTION Relation

SectionID	CourseNumber	Quarter	Instructor
85	CS3320	Summer 99	Wu
88	CS3320	Summer 99	Eagle
56	IS3502	Fall 99	Lundy
42	MA3200	Spring 99	Rasmussen
44	MA3200	Summer 99	Carlos

GRADE Relation

StudentNumber	SectionID	Grade
17	85	A
17	56	B
25	88	B
25	44	C

PREREQUISITE Relation

CourseNumber	PrerequisiteNumber
CS3320	CS3300
IS3502	IS2502
CS3600	MA3200
MA3200	MA1100

Figure 2.1: University Database [Ref. 1]

In relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is called a *domain*. The following subsections will define these terms more precisely.

a. Domains, Tuples, Attributes, and Relations

A domain is a set of atomic values. By atomic, we mean that each value in the domain is indivisible as far as the relational model is concerned. A common method of specifying a domain is to specify a data type from which the data values forming the domain are drawn. It is also useful to specify a name for the domain, to help in interpreting its values.

- United States (USA) Phone Numbers: The set of 10-digit phone numbers valid in the United States.
- Social Security Numbers: The set of valid 9-digit social security numbers.
- Grade Point Averages: Possible mean values of computed grade point averages; each must be a value between 0 and 4.
- Employee Ages: Possible ages of employees of a Company; each must be a value between 16 and 80 years.

The preceding are logical definitions of domains. A data type or format is also specified for each domain. For example, the data type for the domain US phone numbers can be declared as a character string of the form (ddd) ddd-dddd, where each d is a numeric (decimal) digit and the first three digits form a valid telephone area code. A domain is thus given a name, data type, and format.

A relation schema R , denoted by $R(A_1, A_2, \dots, A_n)$, is made up of a relation called R and a list of attributes A_1, A_2, \dots, A_n . A relation schema is used to

describe a relation; R is called the name of this relation. Each attribute A is the name of a role played by some domain D in the relation schema R. P is called the domain of A and is denoted by $\text{dom}(A_i)$. The degree of a relation is the number of attributes n of its relation schema. [Ref. 1]

An example of a relation schema for a relation of degree 7, which describes university students, is the following:

STUDENT (Name, SSN, HomePhone, Address, OfficePhone, Age, GPA)

Figure 2.2: STUDENT Relation

b. Characteristics of Relations

A relation is defined as a set of tuples. Mathematically, elements of a set have no order among them; hence, tuples in a relation do not have any particular order. Tuple ordering is not part of a relation definition, because a relation attempts to represent facts at a logical or abstract level. When a relation is implemented as a file, a physical ordering may be specified on the records of the file. [Ref. 1]

According to the preceding definition of a relation, an n-tuple is an ordered list of n values, so the ordering of values in a tuple - and hence of attributes in a relation schema definition - is important. However, at a logical level, the order of attributes and their values are not really important as long as the correspondence between attributes and values is maintained.

Another property of the relation is that there are no duplicate tuples in a relation. This property follows from the fact that the body of the relation is a

mathematical set (i.e., a set of tuples), and sets in mathematics by definition do not include duplicate elements. An important corollary of this fact is that there is always a primary key. Since tuples are unique, it follows that at least the combination of all attributes of the relation has the uniqueness property. [Ref. 2]

Each value in a tuple is an atomic value; that is, it is not divisible into components within the framework of the relational model. Hence, composite and multivalued attributes are not allowed in a relation. Multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes. [Ref. 1]

The values of some attributes within a particular tuple may be unknown or may not apply to that tuple. A special value, called *null*, is used for these cases. In general, we can have several types of null values, such as "value unknown", "attribute does not apply to this tuple", or "this tuple has no value for this attribute".

As a summary, for a table to be a relation the following must hold: The cells of the table must be single valued (atomic), and neither repeating groups nor arrays are allowed as values. All entries in any column must be of the same kind. Each column must have a unique name, but the order of the columns in the table is insignificant. Finally, no two rows in a table may be identical, and the order of the rows is not important. [Ref. 3]

c. *Types of Relations*

There are three types of relations that can exist in a relational system: Base relations, views, snapshots, query results, intermediate results, and temporary relations.

[Ref. 2]

(1) *Base Relations*: A base relation corresponds to a table whose tuples are physically stored in the database; that is, it is a named, autonomous relation. In other words, base relations are those relations that are sufficiently important that the database designer has decided that it is worth giving them a name and making them a direct part of the database.

(2) *Views*: A view is a named, derived relation that is represented within the system purely by its definition in terms of other named relations. It does not have any separate, distinguishable stored data of its own (unlike a base relation).

(3) *Snapshots*: A snapshot is also a named, derived relation, like a view. Unlike a view, however, a snapshot is real, not virtual. It is represented not only by its definition in terms of other named relations, but also by its own stored data.

(4) *Query Results*: A query result is, as the name implies, simply the final output relation resulting from some specified query. It may or may not be named. Query results have no persistent existence within the database.

(5) *Intermediate Results*: An intermediate result is a relation (typically unnamed) that results from some relational expression that is nested within a larger expression.

(6) *Temporary Relations*: A temporary relation is a named relation, like a base relation or view or snapshot, but unlike a base relation or view or snapshot, it is automatically destroyed at some appropriate.

2. Relational Model Constraints

The various types of constraints that can be specified on a relational database schema include domain constraints, key constraints, entity integrity, and referential integrity constraints. Other types of constraints, called *data dependencies* (which include functional dependencies and multivalued dependencies), are used mainly for database design by normalization and will be discussed in Section D of this chapter.

a. Domain Constraints

Domain constraints specify that the value of each attribute "A" must be an atomic value from the domain $\text{dom}(A)$ for that attribute. The data types associated with domains typically include standard numeric data types for integers (such as short-integer, integer, long-integer) and real numbers (float and double-precision float). Characters, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type

where all possible values are explicitly listed. [Ref. 1]

b. Key Constraints

A relation is defined as a set of tuples. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for all their attributes. Usually, there are other subsets of attributes of a relation schema R with the property that no two tuples in any relation instance r of R should have the same combination of values for these attributes. Any such set of attributes is called a *superkey* of the relation schema R . Every relation has at least one superkey -- the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. Hence, a key is a minimal superkey; a superkey from which we cannot remove any attributes and still have the uniqueness constraint hold.

For example, consider the STUDENT relation of Figure 2.2. The attribute set {SSN} is a key of STUDENT, because no two-student tuples can have the same value for SSN. Any set of attributes that includes SSN -- for example {SSN, Name, Age} -- is a superkey.

The value of a key attribute can be used to identify uniquely a tuple in the relation. For example, the SSN identifies uniquely each tuple in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on every relation instance of the schema. A key is determined from the meaning of the attributes in the relation schema.

In general, a relation schema may have more than one key. In this case, each of the keys is called a *candidate key*. It is common to designate one of the candidate keys as the *primary key* of the relation. This is the candidate key whose values are used to identify tuples in the relation.

c. Entity Integrity Constraint

The entity integrity constraint states that no primary key value can be null. This is because the primary key value is used to identify individual tuples in a relation; having null values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had null for their SSN values in the STUDENT relation of Figure 2.2, we might not be able to distinguish them.

d. Referential Integrity Constraint

Key constraints and entity integrity constraints are specified on individual relations. The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. For example, in Figure 2.1, the attribute StudentNumber of GRADE relation stores the student number for which the grade is recorded; hence, its value in every GRADE tuple must match the StudentNumber value of some tuple in the STUDENT relation.

To define referential integrity more formally, we must first define the concept of a *foreign key*. When the key of one relation is stored in a second relation, it is

called a *foreign key*. The attributes in the foreign key must have the same domain as the primary key attributes and the foreign key is said to reference or refer to a second relation.

Referential integrity constraints typically arise from the relationships among the entities represented by the relation schemas. Notice that a foreign key can refer to its own relation. For example, the attribute SUPERSSN in EMPLOYEE relation refers to the supervisor of an employee, which is another employee represented by a tuple in the EMPLOYEE relation. Hence, SUPERSSN is a foreign key that references the EMPLOYEE relation itself.

3. Update Operations on Relations

There are three basic update operations on relations: *insert*, *delete*, and *modify*. Insert is used to add a new tuple or tuples in a relation; delete is used to remove tuples; and modify is used to change the values of some attributes. Whenever update operations are applied, the integrity constraints specified on the relational database schema should not be violated. [Ref. 1]

a. Insert Operation

The insert operation provides a list of attribute values for a new tuple that is to be inserted into a relation. Insert can violate any of the four types of constraints discussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain. Key constraints can be violated if a key value in the new tuple already exists in another tuple in the relation.

Entity integrity can be violated if the primary key of the new tuple is null. Referential integrity can be violated if the value of any foreign key refers to a tuple that does not exist in the referenced relation. [Ref. 1]

If an insertion violates one or more constraints, two options are available. The first option is to reject the insertion. The second option is to attempt to correct the reason for rejecting the insertion.

b. Delete Operation

This operation is used to remove the specified tuples from a relation. The delete operation can violate only referential integrity, if the tuple being deleted is referenced by the foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple to be deleted. [Ref. 1]

Three options are available if a deletion operation causes a violation. The first option is to reject the deletion. The second option is to attempt to cascade (or propagate) the deletion by deleting tuples that reference the tuple that is being deleted. A third option is to modify the referencing attribute values that cause the violation; each such value is either set to null or changed to reference another valid tuple.

c. Modify Operation

The modify operation is used to change the values of one or more attributes in a tuple (or tuples) of a relation. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. [Ref. 1]

Modifying an attribute that is neither a primary key nor a foreign key usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to deleting one tuple and inserting another in its place, because we use the primary key to identify tuples.

4. Relational Algebra

The *relational algebra* is a collection of operations that are used to manipulate entire relations. These operations are used to select tuples from individual relations and to combine related tuples from several relations for the purpose of specifying a query on the database. The result of each operation is a new relation, which can be further manipulated. Relational algebra is closed, which means that the results of one or more relational operations are always in a relational state.

The relational algebra operations are usually divided into two groups. One group includes set operations from mathematical set theory; these are applicable because each relation is defined to be a set of tuples. Set operations include UNION, INTERSECTION, DIFFERENCE, and CARTESIAN PRODUCT. The other group consists of operations developed specifically for relational databases; these include SELECT, PROJECT, and JOIN.

a. Set Operations

Set theoretic operations apply to the relational model, because a relation is defined to be a set of tuples and can be used to process the tuples in two relations as sets.

Several set theoretic operations are used to merge the elements of two sets in various ways, including UNION, INTERSECTION, and DIFFERENCE. These operations are binary; that is, they are applied to two sets. In order to apply any of these three operations on the relational model, it is necessary that the relations have the same type of tuples; this condition is called *union compatibility*.

Two relations are said to be union compatible if they have the same number of attributes and that each pair of corresponding attributes have the same domain.

We can define the three operations UNION, INTERSECTION, and DIFFERENCE on two union-compatible relations, "R" and "S", as follows:

(1) *Union*: The result of this operation is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

(2) *Intersection*: The result of this operation is a relation that includes all tuples that are in both R and S.

(3) *Difference*: The result of this operation is a relation that includes all tuples that are in R but not in S.

(4) *Cartesian Product*: The result of this operation is a relation that includes one tuple for each combination of tuples – one from R and one from S; that

is every tuple from R is combined with every tuple from S. The relations on which CARTESIAN PRODUCT operation is applied do not have to be union compatible.

b. SELECT Operation

The SELECT operation is used to select a subset of the tuples in a relation that satisfy a selection condition. In general, the SELECT operation is denoted by

$$\sigma_{\langle \text{selection condition} \rangle} (\langle \text{relation name} \rangle)$$

The relation resulting from the SELECT operation has the same attributes as the relation on which this operation is applied. The Boolean expression specified in the selection condition is made up of a number of clauses of the form:

$\langle \text{attribute name} \rangle \langle \text{comparison operator} \rangle \langle \text{constant value} \rangle$, or

$\langle \text{attribute name} \rangle \langle \text{comparison operator} \rangle \langle \text{attribute name} \rangle$

where $\langle \text{attribute name} \rangle$ is the name of an attribute of $\langle \text{relation name} \rangle$, $\langle \text{comparison operator} \rangle$ is one of the operators =, <, ≤, >, ≥, ≠ and $\langle \text{constant value} \rangle$ is a constant value from the attribute domain. Clauses can be connected by the Boolean operators AND, OR, and NOT to form a general condition.

The SELECT operator is unary; that is, it is applied on a single relation. Hence, SELECT cannot be used to select tuples from more than one relation.

c. PROJECT Operation

If one might think of a relation as a table, the SELECT operation selects some of the *rows* from the table while discarding other rows. The PROJECT operation, on the other hand, selects certain *columns* from the table and discards the other columns.

If we are interested in only certain attributes of a relation, we use the PROJECT operation to "project" the relation over these attributes. Projection can also be used to change the order of attributes in a relation. The general form of a PROJECT operation is

$$\pi_{\langle \text{attribute list} \rangle} (\langle \text{relation name} \rangle)$$

where $\langle \text{attribute list} \rangle$ is a list of attributes of the relation specified by $\langle \text{relation name} \rangle$.

The resulting relation has only the attributes specified in $\langle \text{attribute list} \rangle$ and in the same order as they appear in the list. The PROJECT operation implicitly removes any duplicate tuples, so the result of the PROJECT operation is a set of tuples and a valid relation.

d. JOIN Operation

The JOIN operation, denoted by \bowtie , is used to combine related tuples from two relations into single tuples. This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations. Essentially, JOIN operation is the same as a Cartesian Product followed by a SELECT operation. The general form of a JOIN operation on two relations R and S is

$$R \bowtie_{\langle \text{join condition} \rangle} S$$

The resulting relation has one tuple for each combination of tuples whenever the combination satisfies the join condition. The most common JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an Equijoin. In the result of an Equijoin, there are always one or more pairs of attributes that have identical values in every tuple.

Because one of each pair of attributes with identical values is superfluous, a new operation, called *Natural Join*, was created to get rid of the second attribute in an equijoin condition.

C. STRUCTURED QUERY LANGUAGE (SQL)

SQL is a declarative database language designed for use with relational databases. It has been endorsed by the American National Standards Institute (ANSI) as the language for manipulating relational databases, and it is the data access language used by many commercial DBMS products, including DB2, ORACLE, INGRES, SYBASE, SQL Server, dBase, Microsoft Access, Paradox, and many others. Originally, SQL was called Structured English Query Language (SEQUEL) and was designed and implemented at IBM Research as the interface for an experimental relational database system.

SQL is a comprehensive database language; it has statements for data definition, query, and update. Hence, it is both a Data Definition Language (DDL) and a Data Manipulation Language (DML). In addition, it has facilities for defining views on the database, for creating and dropping indexes on the files that represent relations, and for embedding SQL statements into a general purpose programming language, such as C++, Java or Pascal. [Ref. 1]

SQL consists of a set of standard commands that can be understood by all compliant Relational Database Management Systems (RDBMS). The following is a list of more commonly used SQL commands.

1. Data Definition in SQL

SQL uses the terms *table*, *row*, and *column* for relation, tuple, and attribute, respectively. The SQL commands for data definition are CREATE, ALTER, and DROP. These commands are used to create or modify tables and other database objects and are explained in the following subsections.

a. CREATE TABLE Command

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and constraints. The attributes are specified first; and each attribute is given a name, a data type to specify its domain of values, and possibly some constraints. The key, entity integrity, and referential integrity constraints are the specified. [Ref. 1]

The following is the command used to define a table with the name Department:

```
CREATE TABLE Department
  (DepartmentName      VARCHAR(10)      NOT NULL,
  ManagerName         CHAR (15)        NOT NULL,
  DepartmentNumber    INT              NOT NULL,
  PRIMARY KEY (DepartmentNumber),
  FOREIGN KEY (ManagerName) REFERENCES Employee (Name));
```

The data types that are available for attributes in SQL include numeric, character string, bit string, date, and time. Numeric data types include integer numbers of various sizes (INTEGER AND SMALLINT), and real numbers of various precisions (FLOAT, REAL, DOUBLE). Formatted numbers can be declared by using DECIMAL (i, j) or NUMERIC (i,j), where “i” is the total number of decimal digits, and “j” is the number of digits after the decimal point. Character string data types are either fixed-length (CHAR (n), where n is the number of characters) or varying-length (VARCHAR (n), where n is the maximum number of characters). Bit string data types are either of fixed length n (BIT (n)) or varying-length (BIT VARYING (n), where n is the maximum number of bits. [Ref. 1]

b. DROP TABLE Command

This command is used to delete a table definition and all rows in the table. There are two drop behavior options: CASCADE and RESTRICT. For example, if we no longer need to keep track of departments in our database, we can get rid of the Department table by issuing the following command:

DROP TABLE Department CASCADE;

With the CASCADE option, all constraints and views that reference the Department table are dropped automatically from the database schema, along with the table itself. If the RESTRICT option is chosen instead of CASCADE, Department table is deleted only if it is not referenced in any constraints (such as by foreign key definitions in another relation) or views. [Ref. 1]

c. ***ALTER TABLE Command***

The definition of a table can be changed by using the ALTER TABLE command. It is possible to add or drop a column, change a column definition, or add/remove constraints defined for the table. The following example shows the command used to add another column to the Department table defined above.

```
ALTER TABLE Department ADD ManagerStartDate DATE;
```

To drop a column from a table, one must choose either CASCADE or RESTRICT for drop behavior. For example, the following command removes the attribute ManagerName from the Department table.

```
ALTER TABLE Department DROP ManagerName CASCADE;
```

2. Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. This is the most commonly used SQL command and is used to query the database and display selected data to the user. The SELECT statement is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT <attribute list>
```

```
FROM <table list>
```

```
WHERE <condition>
```

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a Boolean search expression that identifies the tuples to be retrieved by the query. [Ref. 1]

The following query retrieves the names and numbers of the departments that are managed by John Lewis.

```
SELECT DepartmentName, DepartmentNumber  
FROM Department  
WHERE ManagerName = 'John Lewis';
```

SQL provides five built-in functions COUNT, SUM, AVG, MAX, and MIN that can be used in query statements. For example, the following query finds the sum of all employees of the 'Research' department, as well as the maximum, minimum, and the average salary in this department.

```
SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)  
FROM Department, Employee  
WHERE DepartmentName = 'John Lewis' AND DNO = DepartmentNumber;
```

It is also possible to apply built-in functions to groups of rows within a table. If we want to apply the aggregate functions to subgroups of tuples in a relation, based on some attribute values, we can use the GROUP BY clause for this purpose. The following query retrieves the department name, the number of employees in the department, and their average salary.

```
SELECT DepartmentName, COUNT (*), AVG (Salary)
FROM Employee
GROUP BY DepartmentName;
```

Finally, SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the ORDER BY clause. For example, suppose one wants to retrieve a list of employees but wants the list ordered by the employees' departments and may want the names within each department ordered alphabetically.

```
SELECT DepartmentName, LastName, FirstName
FROM Employee, Department
WHERE DNO = DepartmentNumber
ORDER BY DepartmentName, LastName, FirstName;
```

3. Update Statements in SQL

In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE.

a. *INSERT Command*

The INSERT command is used to add a new row to a table. The relation name and a list of values for the tuple must be specified. It can be used to fill a new table with data or add new data to an already existing table. When the user of the POET database system performs an operation to input new data into the database, the request is carried out by means of an INSERT command. The example below adds new department to the table defined above.

```
INSERT INTO Department (DepartmentName, ManagerName, DepartmentNumber)
VALUES ("Research", "John Lewis", 32015);
```

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values in the INSERT command. In this case, attributes not specified in the INSERT statement are set to their DEFAULT values or NULL, and the values are listed in the same order as the attributes are listed in the INSERT command itself. [Ref. 1]

b. DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in SQL query, to select the rows to be deleted. Depending on the number of tuples selected by the condition in the WHERE clause, zero, one, or several tuples can be deleted by a single DELETE command. A missing WHERE clause specifies that all rows in the table are to be deleted; however, the relation remains in the database as an empty table. [Ref. 1] The following query can be used to delete the department whose manager is "John Lewis".

```
DELETE FROM Department  
WHERE ManagerName = "John Lewis";
```

c. UPDATE Command

This command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the rows to be modified from a single relation. An additional SET clause specifies the attributes to be modified and their new values. UPDATE queries are used in the POET to modify existing data in the back-end database. The following query can be used, for example, to change the manager name of the department that has a Department Number of 32014.

```
UPDATE Department  
  
SET ManagerName = 'Adam Smith'  
  
WHERE DepartmentNumber = 32014;
```

4. Views in SQL

A view in SQL is a single table that is derived from other base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a virtual table, in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.

A view is a way of specifying a table that is needed to be referenced frequently, even though it may not exist physically. For example, frequent issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the EMPLOYEE, WORKS_ON, and PROJECT tables every time that query is issued, a view can be defined that is a result of these joins and, hence, already includes the attributes to be frequently retrieved.

```
CREATE VIEW EMPLOYEE_PROJECT  
  
AS   SELECT FirstName, LastName, ProjectName, Hours  
  
      FROM Employee, Project, Works_On  
  
      WHERE SSN = ESSN AND PNO = PNumber;
```

The view is specified by the CREATE VIEW command. The view is given a virtual table name, a list of attribute names, and a query to specify the contents of the view. If new attribute names are not specified for the view, as in the above example, then the view inherits the names of the view attributes from the defining tables. [Ref. 1]

5. Processing SQL Statements

SQL statements can result in computationally expensive function calls; for example, a complex join operation between two or more large tables. It is important that system designers have a general understanding of how a database management system processes an SQL statement. To process an SQL statement, a DBMS performs four basic steps: Parse the SQL statement, Validate the statement, Generate an Access Plan, and Execute the Plan. [Ref. 4]

a. Parse the SQL Statement

The DBMS first parses the SQL statement. It breaks the statement up into individual words, called *tokens*, and makes sure that the statement has a valid verb and valid clauses, and so on. Syntax errors and misspellings can be detected in this step. Parsing a SQL statement does not require access to the database and typically can be done very quickly. This phase ensures that the statement is syntactically correct. [Ref. 4]

b. Validate the SQL Statement

The second step in executing a query is validating the statement. The DBMS checks the statement against the system catalog. The system catalog contains database metadata, including table names, attributes and types. This phase ensures the statement parameters are semantically correct. Do all the tables named in the SQL statement exist in the database? Do all of the columns exist and are the column names unambiguous? Does the user have the required privileges to execute the statement? Certain semantic errors can be detected in this step. [Ref. 4]

c. Generate an Access Plan

The DBMS is responsible for managing the data stored in the database. In this phase, based upon the statement, the DBMS generates an access plan. The access plan is a binary representation of the steps that are required to execute the statement. The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.

Optimization is a very CPU-intensive process and requires access to the system catalog. For a complex, multi-table query, the optimizer may explore thousands of different ways of carrying out the same query. However, the cost of executing the query inefficiently is usually so high that the time spent in optimization is regained in increased query execution speed.

This is even more significant if the same optimized access plan can be reused to perform repetitive queries. [Ref. 4]

d. Execute the Access Plan

In this step, the DBMS will execute the access plan, producing a result set that can be passed to the user.

6. SQL Techniques

There are a variety of ways to use SQL to define and manipulate data in a Database Management System. This section will briefly present embedded SQL, stored procedures, and call level interface. The implementation details of the following are usually specific to each DBMS. [Ref. 5]

a. Embedded SQL

One way is to embed SQL statements in a high-level programming language. This is called Embedded SQL. Embedded SQL allows programmers to place SQL statements into a host language, such as Java or C++. The SQL Statement can be static or dynamic. Static SQL is effective if the data access can be determined at program design time and is used when speed is important.

Each SQL statement starts with an introducer and ends with a terminator, which serves as a flag. The code is processed by a SQL pre-compiler (provided by the DBMS vendor), which separates the source code and the SQL request. The pre-compiler substitutes calls to proprietary DBMS routines that provide the run-time link between the

program and the DBMS. The revised source code is then compiled and ultimately linked with the proprietary DBMS library producing the executable. [Ref. 4]

The SQL requests that were extracted from the program form a database request module, which is processed by a binding utility. This utility examines the SQL statements, parses, validates, and optimizes them, and produces an access plan for each statement. Because the SQL Statement is hard coded, this processing only needs to occur at compile time, not at run time, resulting in faster run time query execution.

Dynamic SQL is effective when the data access cannot be determined in advance, such as allowing a user to enter a SQL statement in which the results will be displayed in a grid object. The application uses a flag, such as a question mark as a place holder for parameters that will be supplied later. The SQL statement with the embedded flags is then sent to the DBMS, via a *PREPARE* (string name) method. This allows the DBMS to parse the string, and prepare an access plan. [Ref. 4]

When the user enters the input parameters in the client program the application will call *EXECUTE* (string name), passing the DBMS the valid parameters. The DBMS can then execute the query and provide the result set back to the user. This technique is not as fast as Static Embedded SQL, because of the need to bind the input parameters.

b. Stored Procedures

Another way to execute SQL statements is to have pre-defined and compiled procedures, which reside on the database and can be called by clients. These procedures are commonly referred to as stored procedures. A stored procedure is pre-compiled SQL code that resides on the database server. Stored procedures take input parameters and return a result. A number of procedures can be packaged to form an SQL Module, which can be stored in the DBMS or linked to the application. A module provides logical separation of SQL statements and the programming language/statements.

[Ref. 4]

Stored procedures are a form of query optimization. They are used for efficiency, for SQL statements that are frequently executed and are computationally expensive. The database administrator (DBA) creates and stores the procedures in the DBMS. Once they are stored, those procedures can be invoked by a client. So, instead of submitting a SQL statement, the client will simply invoke the stored procedure the DBA has already defined. The database management system can develop, optimize and store an access plan for executing the stored procedure, therefore decreasing response time when the client invokes the procedure, because the access plan will not have to be regenerated. Stored procedures can also reduce network congestion, by returning only the result set of an operation rather than entire tables that the client may further process.

Triggers are another form of stored procedures. Triggers are special, user defined actions in the form of a stored procedure, that are automatically invoked by the server based upon data related events.

An example of a trigger might be the automatic generation of a parts order if the inventory level of widgets falls below a certain level. [Ref. 4]

The problem with stored procedures is that they are vendor-specific, totally non-standard, not portable across platforms, and have no standard interface definition language or stub compiler. Therefore, there is no standard way to pass or define parameters. [Ref. 5]

c. Call Level Interface

Another alternative to Embedded SQL is to use a callable SQL Application Programming Interface (API) for database access, providing the application with a library of DBMS functions that can be called by the application program. The database aware application calls CLI functions on the local system, and the calls are sent across the network and processed by the DBMS. The initial call may be to establish a connection with the remote database. The application builds its SQL statements, places the statement in a buffer then makes a call to send the statement to the DBMS for processing. Then the application makes a CLI call to disconnect from the DBMS. [Ref. 4]

An API does not require a pre-compiler to convert SQL statements into a high-level language, which can then be compiled and executed on the database. Instead, an API allows the user to create and execute SQL statements at run time. A standard API can be used to produce portable applications that are independent of any database product. The SQL Access Group Call Level Interface (SAG CLI) specifies a common API for accessing multiple databases. It provides common SQL semantics and syntax, codifies the SQL data types, and provides common error handling and reporting.

The SAG API enables the client to connect to a database, execute requests, retrieve the results and terminate the connection. Table 3.2 provides a comparison of the features of CLI and Embedded SQL. [Ref. 5]

Microsoft's ODBC is a Windows API that is an extended version of the SAG CLI. In addition to its basic functionality, it provides methods to retrieve information about the database and handle multimedia types of data. ODBC offers the ability to connect to multiple kinds of databases on different platforms. However, the following are its drawbacks: [Ref. 5]

- It is procedure oriented and thus does not mold with most of the application programs written in an object-oriented language.
- ODBC standards are controlled by one vendor and are subject to change(s) at the vendor's wish.
- ODBC is difficult to learn and debug. It mixes simple and advanced features together.
- ODBC driver manager and drivers must be installed on every client machine. This means it might be a poor choice for a web-based database system.
- ODBC has drawbacks in the security, robustness and portability of applications.

Because of these drawbacks, and since Java is the natural language of choice for an Internet based database system, JDBC was developed by Sun Microsystems as a high-level API for invoking SQL commands directly on different vendor databases. JDBC provides the security, robustness and portability that ODBC lacks. JDBC is a Java API that enables large-scale applications to provide pure Java solutions. Java and JDBC will be explained in Chapter IV.

D. NORMALIZATION

Relational database tables sometimes suffer from some rather serious problems in terms of performance, integrity, and maintainability. For example, when the entire database is defined as a single large table, it can result in a large amount of redundant data and lengthy searches for just a small number of target rows. It can also result in long and expensive updates and deletions in particular can result in the elimination of useful data as an unwanted side effect.

If we had a method of breaking up such a large table into smaller tables so that these types of problems would be eliminated, the database would be much more efficient and reliable. Classes of relational database schemes or table definitions, called *normal forms*, are commonly used to accomplish this goal. The creation of a normal form database table is called *normalization*. It is accomplished by analyzing the interdependencies among individual attributes associated with those tables and taking projections (subsets of columns) of larger tables to form smaller ones. [Ref. 6]

Normalization of data can be looked on as a process during which unsatisfactory relation schemas are decomposed by breaking up their attributes into smaller relation schemas that possess desirable properties. The normalization process, as first proposed by E. F. Codd in 1972, takes a relation schema through a series of tests to certify whether or not it belongs to a certain normal form. [Ref. 1]

To understand the normalization, it is important to define three important terms, *functional dependency*, *key*, and *update anomaly*.

1. Functional Dependencies

A *functional dependency* is a relationship between or among attributes. In a situation given the value of one attribute, one can obtain the value of another attribute. For example, if we know the value of Social Security Number, we can find the value of the Employee Name. If this is true, it is surmised that EmployeeName is *functionally dependent* on SocialSecurityNumber or SocialSecurityNumber *functionally* (or uniquely) *determines* EmployeeName. [Ref. 3]

In more general terms, attribute Y is functionally dependent on attribute X, if the value of X uniquely determines the value of Y. The functional dependency between X and Y is denoted by the notation $X \rightarrow Y$. The attributes on the left side of the arrow are called *determinants*. [Ref. 1]

A functional dependency is a property of the meaning or semantics of the attributes. We use our understanding of the semantics of the attributes of a relation, that is

how they relate to one another, to specify the functional dependencies that should hold on all relation states. Consider the following EMPLOYEE-PROJECT relation schema:

**EMPLOYEE-PROJECT (SSN, ProjectNo, Hours, EmployeeName, ProjectName,
ProjectLoaction)**

From the semantics of the attributes, we know that the following functional dependencies should hold:

(a) $SSN \rightarrow EmployeeName$

(b) $ProjectNo \rightarrow \{ProjectName, ProjectLocation\}$

(c) $\{SSN, ProjectNo\} \rightarrow Hours$

These functional dependencies specify that (a) the value of an employee's social security number (SSN) uniquely determines the employee's name (EmployeeName); (b) the value of a project's number (ProjectNo) uniquely determines the project name (ProjectName) and location (ProjectLocation); and (c) a combination of SSN and ProjectNo values uniquely determines the number of hours the employee works on the project per week (Hours).

2. Keys

A *key* is a group of one or more attributes that uniquely identifies each row in a relation. A key is determined from the meaning of the attributes in the relation schema. A set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on every relation instance of the schema. Consider the following ACTIVITY relation schema:

ACTIVITY (StudentID, Activity, Fee)

The meaning of a row is that a student engages in the named activity for the specified fee. Assume that a student is allowed to participate in only one activity at a time. In this case, a value of StudentID determines a unique row, and so it is a key. [Ref. 3]

Keys can also be composed of a group of attributes taken together. For example, if students were allowed to enroll in many activities at the same time, it would be possible for one value of StudentID to appear in two or more rows of the table, so StudentID could not uniquely identify the row. In this case, the combination of (StudentID, Activity) can uniquely identify each row.

In general, a relation schema may have more than one key. In this case, each of the keys is called a *candidate key*. It is common to designate one of the candidate keys as the *primary key* of the relation. This is the candidate key whose values are used to identify tuples in the relation. [Ref. 1]

3. Update Anomalies

A table that meets the minimum definition of a relation may not have an effective or appropriate structure. For some relations, changing the data can have undesirable consequences, called *update anomalies*. These can be classified into insertion anomalies, and modification anomalies. Anomalies can be eliminated by redefining the relation into two or more relations. [Ref. 3]

a. Insertion Anomalies

Consider the ACTIVITY relation that is shown in Figure 2.1. Suppose we want to store the fact that scuba diving costs \$175, but we can not enter this data into the ACTIVITY relation until a student takes up scuba diving. This restriction is called an *insertion anomaly*. A fact about one entity can not be inserted until we have an additional fact about another entity.

StudentID	Activity	Fee
100	Skiing	200
150	Swimming	50
175	Squash	50
200	Swimming	50

Figure 2.3: Activity Relation [Ref. 3]

b. Deletion Anomalies

If the tuple is deleted for Student 100 from the ACTIVITY relation shown in Figure 2.1, it might be lost that Student 100 is a skier, but also the fact that skiing costs \$200. This is called a *deletion anomaly*; that is, by deleting the facts about one entity, one might inadvertently delete facts about another entity. With one deletion, facts about two entities might be lost.

c. Modification Anomalies

In the ACTIVITY relation shown in Figure 2.1, if the value of fee attribute of a particular activity is changed -- for example, the fee for swimming is \$75 -- one must update the tuples of all students who enroll in that activity; otherwise, the database will become inconsistent. A failure to update some records, the same activity will be shown to have different fees for different students, which should not be the case. [Ref. 1]

One can eliminate the insertion, deletion and the modification anomalies by dividing the ACTIVITY relation into two relations, each one dealing with a different theme. For example, the StudentID and Activity attributes can be put into one relation, called STUDENT-ACTIVITY and the Activity and Fee attributes can be put into another relation called ACTIVITY-COST. Now, if Student 100 is deleted from STUDENT-ACTIVITY, the fact that skiing costs \$200 is not lost. Furthermore, scuba diving can be added and its fee to the ACTIVITY-COST relation before the student enrolls. Also, one update in the ACTIVITY-COST relation will be sufficient to change the fee of the swimming activity. Thus, the insertion, deletion and the modification anomalies have been eliminated.

4. Normal Forms

Relations can be classified by the types of update anomalies to which they are vulnerable. These classes of relations and the techniques for preventing anomalies are called *normal forms*.

Initially, Codd proposed three normal forms in 1970, which he called first, second, and third normal form. A stronger definition of 3NF was proposed later by Boyce and Codd and is known as Boyce-Codd normal form (BCNF). Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multi-valued dependencies. [Ref. 1]

These normal forms are nested; that is, a relation in second normal form is also in first normal form, and a relation in 5NF is also in 4NF, BCNF, 3NF, 2NF and 1NF. A serious limitation of these normal forms was that no theory guaranteed that any of them would eliminate all anomalies; each form could eliminate just certain ones. This changed, however, in 1981 when R. Fagin defined a new normal form called *domain/key normal form* (DK/NF). Fagin showed that a relation in domain/key normal form is free of all modification anomalies, regardless of their type and that any relation that is free of modification anomalies must be in domain/key normal form. [Ref. 3]

a. *First Normal Form (1NF)*

First normal form was defined to disallow multi-valued attributes, composite attributes, and their combinations. It states that the domains of the attributes must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute. Hence, 1NF does not

allow having a set of values, a tuple of values, or a combination of both as an attribute value for a single tuple. The only attribute values permitted by 1NF are single atomic values. [Ref. 1]

1NF is now considered to be part of the formal definition of a relation. Any table of data that meets the definition of a relation is said to be in *first normal form*. For a table to be a relation, the following rules must hold: [Ref. 3]

- The cells of the table must be single valued, and neither repeating groups nor arrays are allowed as values.
- All entries in any column (attribute) must be of the same kind.
- Each column must have a unique name, but the order of the columns in the table is insignificant.
- No two rows in a table may be identical, and the order of the rows is not important.

The advantages of 1NF over unnormalized tables are its representational simplicity and the ease with which one can develop a query language for it. The disadvantage is the requirement of duplicate data. [Ref. 6]

b. Second Normal Form (2NF)

Second normal form is based on the concept of full functional dependency. A functional dependency $X \rightarrow Y$ is a full functional dependency if removal of any attribute from X means that the dependency does not hold any more. A functional

dependency $X \twoheadrightarrow Y$ is a partial dependency if some attribute can be removed from X and the dependency still holds. [Ref. 1]

EMPLOYEE-PROJECT (SSN, ProjectNo, Hours, EmployeeName, ProjectName, ProjectLoaction)

In the above EMPLOYEE-PROJECT relation, $\{SSN, ProjectNo\} \twoheadrightarrow Hours$ is a full functional dependency, because neither $SSN \twoheadrightarrow Hours$ nor $ProjectNo \twoheadrightarrow Hours$ holds. However, the dependency $\{SSN, ProjectNo\} \twoheadrightarrow EmployeeName$ is a partial dependency, because $EmployeeName$ is dependent on only SSN .

A table is in *second normal form (2NF)* if and only if it is in 1NF and every non-key attribute is fully dependent on the primary key [Ref. 6]. According to this definition, if a relation has a single attribute as its key, then it is automatically in second normal form. Since the key is only one attribute, by default, every non-key attribute is dependent on the primary key; there can be no partial dependencies. Thus, second normal form is of concern only in relations that have composite keys.

If a relation schema is not in 2NF, it can be further normalized into a number of 2NF relations in which non-key attributes are associated only with the part of the primary key on which they are fully functionally dependent.

c. Third Normal Form (3NF)

The tables in 2NF represent a significant improvement over 1NF tables; however, they still suffer from the anomalies, but for different reasons associated with transitive dependencies. If a transitive dependency exists in a table, it means that two

separate facts are represented in that table, one fact for each functional dependency involving a different left side. [Ref. 6] Consider the following HOUSING relation schema:

HOUSING (StudentID, Building, Rent)

The primary key of this relation is StudentID, and the functional dependencies are StudentID \rightarrow Building and Building \rightarrow Rent. These dependencies arise because each student lives in only one building and each building charges only one rent. Since StudentID determines Building and Building determines the value of Rent, then StudentID indirectly determines the Rent. An arrangement of functional dependencies like this is called a *transitive dependency*, since StudentID determines Rent through the attribute Building.

To eliminate the anomalies from a relation in 2NF, the transitive dependency must be removed, which leads to a definition of 3NF: A relation is in third normal form if it is in second normal form and has no transitive dependencies. [Ref. 3]

Third normal form, which eliminates most of the anomalies known in databases today, is the most common standard for normalization in commercial databases. The few remaining anomalies can be eliminated by the Boyce-Codd normal form and higher normal forms, which will be defined in the following subsections.

d. Boyce-Codd Normal Form (BCNF)

Boyce-Codd normal form is a stronger form of normalization than 3NF, because it does not allow the right side of the functional dependency to be a candidate key. Thus, every left side of a functional dependency in a table must be a candidate key.

[Ref. 6]

A relation is in Boyce-Codd normal form, if the determinants in each of the functional dependencies are candidate keys. Relations in BCNF have no anomalies in regard to functional dependencies.

e. Fourth Normal Form (4NF)

Fourth normal form is related with the concept of *multivalued dependency*. Multivalued dependencies are a consequence of first normal form, which disallowed an attribute in a tuple to have a set of values. If we have two or more multivalued independent attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation instances consistent. This constraint is specified by a multivalued dependency. In other words, whenever two independent one to many relationships are mixed in the same relation, a multivalued dependency may arise. [Ref. 1]

In general, a multivalued dependency exists when a relation has at least three attributes, two of them are multivalued and their values depend on only the third attribute. For example, consider the following relation:

EMPLOYEE (EmployeeName, ProjectName, Dependent)

A tuple in this EMPLOYEE relation represents the fact that an employee may work on several projects and may have several dependents, and the employees, projects and dependents are not directly related to one another. To keep the tuples in the relation consistent, we must keep a tuple to represent every combination of an employee's dependent and an employee's project.

The definition of 4NF, which is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations. A relation schema is in 4NF, if it is in BCNF and has no multivalued dependency.

f. Fifth Normal Form (5NF)

Fifth normal form is based on the concept of *join dependency* and *lossless decomposition*. A table is in fifth normal form if it can not have a lossless decomposition by the projection operation into any number of smaller tables. [Ref. 6]

A lossless decomposition of a table implies that it can be decomposed by two or more projections, followed by a natural join of those projections (in any order) that results in the original table, without any spurious or missing rows. The general lossless decomposition constraint, involving any number of projections, is also known as a *join dependency*. In other words, a table is not in 5NF if it can be lossless decomposed/joined via some projections. [Ref. 1]

If a table is already 4NF, with at least some of the functional dependencies preserved, then the most appropriate decomposition to 5NF is by candidate key, with each smaller table having the candidate key replicated and one non-key associated with the candidate key. If there is only one candidate key - the composite of all attributes -

further decomposition is accomplished by trial and error using various (more than two) subsets of the table's attributes.

Discovering join dependencies in practical databases with hundreds of attributes is difficult; hence, current practice of database design pays scant attention to them. [Ref. 1]

g. Domain-Key Normal Form (DKNF)

The idea behind domain-key normal form is to specify, (theoretically, at least) the "ultimate normal form" that takes into account all possible types of dependencies and constraints. A relation is said to be in DKNF if all constraints and dependencies that should hold on the relation can be enforced simply by enforcing the domain constraints and the key constraints specified on the relation. For a relation in DKNF, it becomes very straightforward to enforce the constraints by simply checking that each attribute value in a tuple is of the appropriate domain and that every key constraint on the relation is enforced. However, it seems unlikely that complex constraints can be included in a DKNF table; hence, its practical utility is limited. [Ref. 1]

In 1981, R. Fagin showed that a relation in domain/key normal form has no modification anomalies, and furthermore, that a relation having no modification anomalies must be in domain-key normal form. This finding establishes a bound on the definition of normal forms, and so no higher normal form is needed, at least in order to eliminate modification anomalies. Equally important, DKNF involves only the concepts of key and domain, concepts that are fundamental and close to the heart of database practitioners. Informally, a relation is in DKNF if enforcing key and domain constraints

causes all of the constraints to be met. Moreover, since relations in DKNF can not have modification anomalies, the DBMS can prohibit them by enforcing key and domain restrictions. [Ref. 3]

5. Summary

Normalization is a process in which larger tables are decomposed to form smaller ones in order to eliminate update anomalies. It is accomplished by analyzing the interdependencies among individual attributes associated with those tables and taking projections of larger tables to create smaller ones.

In some cases, normalization may not be desirable. Whenever a table is spilt into two or more tables, referential integrity constraints are created. If the cost of the extra processing of the two tables and their integrity constraint is greater than the benefit of avoiding modification anomalies, then normalization is not recommended.

Database designers need not normalize the tables to the highest normal form. Relations may be left in lower normal forms for performance reasons. In some cases, creating repeating columns is preferred to the standard normalization techniques.

E. ACCESS 97

Microsoft Access is essentially a database management system. Like other products in this category, Access stores and retrieves data, presents information, and automates repetitive tasks. It is used to create, control, and manipulate one of the most common forms of information system: a database. A database system is a collection of integrated information that describes a particular object. Microsoft Access is a very flexible program that can be used to manage simple database applications or to build complex corporate management information systems.

Microsoft Access is well suited for both creating new database systems and for expanding or upgrading current systems. Microsoft Access can accept data from a wide variety of file formats, which makes it ideal for converting data stored in a different system. There is complete interoperability between Access and Word, Excel, and PowerPoint. Also, the program has an easy-to-master graphical interface, which makes it an ideal tool for less experienced users. [Ref. 7]

Using Object Linking and Embedding (OLE) objects in Windows 95/98 and Microsoft Office 97 products (Excel, Word, PowerPoint, and Outlook), one can extend Access into being a true database operating environment through integration with these products. With the new Internet extensions, it is possible to create forms that interact with data directly from the World Wide Web and translate the forms directly into HTML that works with products like Microsoft Internet Explorer and Netscape Navigator. [Ref. 8]

Even so, Access is more than just a database manager. As a relational database management system, it gives the user access to all types of data and makes it possible to use more than one database table at a time. One can link an Access table with mainframe

or server data or use a table created in Paradox or dBase. The user can take the results of the link and combine the data with an Excel worksheet quickly and easily. [Ref. 8]

At the lowest level, Access gives the end user the capability of creating *tables*, *queries*, *forms*, and *reports* easily. One can perform simple processing by using *expressions*, also known as functions, to validate data, enforce a business rule, or display a number with a currency symbol. *Macros* allow for automation without programming, whereas VBA (Visual Basic for Applications) code lets the user program complex processes. Finally, by using Windows API calls to functions or DLLs written in other languages such as C++, Java, or even Visual Basic, a programmer can write interfaces to other programs and data sources. [Ref. 8]

Access is a set of tools for end-user database management. Access has a table creator, a form designer, a query manager, and a report writer. Access is also an environment for developing applications. By using macros or modules to automate tasks, it is possible to create user-oriented applications as powerful as those created with programming languages - complete with the buttons, menus, and dialog boxes. [Ref. 8]

1. Features of Access 97

The following paragraphs will briefly describe some key features in Access 97: True Relational Database Management; Ease-of-Use Wizards and Builders; Importing, Exporting, and Linking External Files; Powerful Forms and Reports; Multiple-Table Queries and Relationships; Business Graphs and Charts; DDE and OLE Capabilities; Built-in Functions; and Context-Sensitive Help and Office Assistant.

a. True Relational Database Management

Access provides true *relational database management*. Access includes definitions for primary and foreign keys and has full referential integrity built in at the level of the database engine itself, which prevents inconsistent updates or deletions. In addition, tables in Access have data-validation rules to prevent inaccurate data regardless of how data is entered, and every field in a table has format and default definitions for more productive data entry. Access supports all the necessary field types, including Text, Number, AutoNumber (counter), Currency, Date/Time, Memo, Yes/No, Hyperlink, and OLE objects. When values are missing in special processing, Access provides full support for null values.

The relational processing in Access fills many needs with its flexible architecture. It can be used as a stand-alone database management system, in a file-server configuration, or as a front-end client to products such as an SQL server. In addition, Access features ODBC (Open Database Connectivity) that makes it possible to connect to many more external formats.

The program provides complete support for transaction processing, ensuring the integrity of transactions. In addition, user level security provides control over assigning user and group permissions to view and modify database objects. [Ref. 8]

b. Ease-of-Use Wizards and Builders

A *Wizard* can turn hours of work into minutes. Wizards ask the user questions about content, style, and format; then they build the object for you automatically. Access features nearly 100 Wizards to design databases, applications,

tables, forms, reports, graphs, mailing labels, controls, and properties. The user can even customize Wizards for use in a variety of tasks. [Ref. 8]

In some areas, such as programming buttons on forms and reports, wizards make the core so easy that even a fairly naïve Access user can make applications that work just like the ones done by experts. There is nothing one can do with a wizard that he can not do manually, but using wizards can save a lot of time. [Ref. 7]

c. Importing, Exporting, and Linking External Files

Access lets the user import from or export to many common formats, including dBase, FoxPro, Excel, SQL Server, Oracle, Btrieve, many ASCII text formats (including fixed width and delimited), as well as data in HTML format. Importing creates an Access table; exporting an Access table creates a file in the native file format that is being exported to.

Linking (formally known as attaching) means that one can simply use external data without creating an Access table. One can link to dBase, FoxPro, Excel, ASCII, and SQL data. Linking to external tables and then relating them to other tables is a powerful capability. [Ref. 8]

d. Powerful Forms and Reports

The Form and Report Design windows share a common interface and power. The user can add labels, text data fields, option buttons, tab controls, check boxes, lines, boxes, colors, shading -- even pictures, graphs, subforms, or subreports -- to the forms and reports. As the user add each control, he can see the form take shape as he

builds the design. In addition, he has complete control over the style and presentation of data in a form or report. In Access, forms can have multiple pages and reports can have many levels of groupings and totals. The user can also view the report with sample data when he is in design mode so that he does not waste valuable time waiting for a large data file to be processed. [Ref. 8]

Most important, the Report Writer is very powerful, allowing up to ten levels of aggregation and sorting. The Report Writer performs two passes on the data; one can create reports that show the row percentage of a group total, which can be done only by having a calculation based on a calculation that requires two passes through the data.

e. Multiple-Table Queries and Relationships

One of the most powerful features in Access 97 is also the most important. The relationship lets the user link his tables graphically. The user can even link tables of different file types (such as an Access table and a dBase table); when linked, these tables act as a single entity that he can query about his data. It is possible to select specific fields, define sorting orders, create calculated expressions, and enter criteria to select desired records. The user can display the results of a query in a datasheet, form, or report. [Ref. 8]

Queries have other uses as well. One can create queries that calculate totals, display cross-tabulations, and then make new tables from the results. The user can even use a query to update data in tables, delete records, or append one table to another.

f. Business Graphs and Charts

Access 97 has the same graph application found in Microsoft Word, Excel, PowerPoint, and Project. The user can create hundreds of types of business graphs and customize the display to meet his every business need. He can create bar charts, column charts, line charts, pie charts, and area charts in two and three dimensions.

It is possible to add free-form text, change the gridlines, adjust the color and pattern in a bar, display data values on a bar or pie slice, and even rotate the viewing angle of a chart from within the Access Graph program. In addition, the user can link his graph with a form to get a powerful graphic data display that changes from record to record in the table. [Ref. 8]

g. DDE and OLE Capabilities

Through the capabilities of Dynamic Data Exchange (DDE) and Object Linking and Embedding (OLE), the user can add exciting new objects to his Access forms and reports. Such objects may be sound, pictures, graphs, and even video clips. He can easily embed OLE objects (such as a bitmap picture) or documents from word processors (such as Word or WordPerfect) or link to a range of cells in an Excel spreadsheet. By linking these objects to records in his tables, the user can create dynamic database forms and reports and share information between Windows applications.

h. Accessing the Internet

Access 97 has features that allow the user to easily make his applications Internet/intranet ready. It is very easy to save tables, queries, reports, and form datasheets as HTML. The Publish to the Web Wizard allows even a neophyte to place the HTML code generated from an object out on a Web site, ready for the perusal of all who surf the Internet. The Publish to the Web Wizard walks you through the steps of creating the HTML for selected database objects and of placing the generated HTML out on your Web site. Hyperlinks allow others to access the published data as hypertext links, directly from the Access forms. Using the Wizard, it is possible to create either static or dynamic publications, publish them to the Web, create a home page, and even use templates to obtain a standard look and feel for all HTML publications. [Ref. 8]

i. Built-in Functions

Access contains more than 100 functions (small built-in programs that return a value). These functions perform tasks in a wide variety of categories. Access includes database, mathematics, business, financial, date, time, and string functions. The user can use them to create calculated expressions in his forms, reports, and queries.

j. Context-Sensitive Help and Office Assistant

Access provides context-sensitive help; the user can press the F1 key whenever he is stuck. Help information about the item he is working on appears instantly. Access also has an easy-to-use table of contents, a search facility, a history log, and bookmarks.

The Office Assistant is Microsoft's attempt to incorporate artificial intelligence into its help systems. Office Assistant responds in plain English when the user asks for help. Screen Tips, also known as What is This, give short, on-screen explanations of what something is. [Ref. 7]

2. Requirements for Access 97

Access 97 requires specific hardware and software to run. The following subsections will describe the hardware and software requirements needed for Access 97.

a. Hardware Requirements

To use Access 97 successfully, one will need an IBM compatible personal computer (PC) with an 80486SX-33 or higher processor and 12MB of RAM. To get reasonable performance from Access 97, an 80486DX-66 computer with at least 16MB of RAM is recommended. With more memory, the user will be able to run more applications simultaneously, and overall performance will be increased. A fast video card is also recommended to display pictures and graphs.

The user will also need between 60MB and 191MB of hard disk space for a typical installation of Microsoft Office 97. If the user is installing only Access 97, he or she will still need about 50MB, because many of the Office shared files are used by Access and are loaded in the stand-alone version.

Access needs a VGA monitor as a minimum requirement, but an SVGA (or better) display is recommended. This configuration allows the user to view more information at one time and to get a sharper resolution.

A mouse or some other compatible pointing device (trackballs and pens will work) is mandatory to be able to use Access 97. [Ref. 8]

b. Software Requirements

Access requires that Microsoft Windows 95/98 or Windows NT be installed on the computer. Microsoft Office 97 does not run on OS/2 or Windows 3.1.

3. Database Objects and Views in Access 97

The Access database contains six objects; these consist of the data and tools one needs to use Access:

- **Table:** Used to hold the actual data (uses a datasheet to display the raw data)
- **Query:** Used to search, sort, and retrieve specific data
- **Form:** Used to enter and display data in a customized format
- **Report:** Used to display and print formatted data, including calculations and totals
- **Macro:** Used to automate tasks without programming via easy-to-use commands
- **Module:** Program written in VBA

The following subsections will explain each one of these database objects in detail.

a. Tables

A table is a container for raw data. When the user enters data in Access, a table stores it in logical groupings of similar data. The table's design organizes the information into rows and columns. A database contains one or more tables. Most applications in Access have several related tables to present the information efficiently.

Multiple tables simplify data entry and reporting by decreasing the input of redundant data. By defining two tables for an application that uses customer information, for example, one does not need to store the customer's name and address every time the customer purchases an item. By separating the data into multiple tables within the database, the system is easier to maintain, because all records of a given type are within the same table. [Ref. 8]

Datasheets are one of the many ways to view data. Although not a database object, a datasheet displays a list of records from the table in a format commonly known as a browse screen, or table view. A datasheet displays data as a series of rows and columns.

b. Queries

Queries are used to extract information from a database. A query can select and define a group of records that fulfill a certain condition. One can use queries before printing a report so that only the desired data is printed. Forms can also use a query so that only certain records that meet the desired criteria will appear on-screen. The

user can use queries within procedures that change, add, or delete database records.

Access uses the method Query By Example (QBE), to execute the queries. In this method, the user first selects the tables that will be used in the query. When the user enters instructions into the QBE window, Access translates them into SQL statements and retrieves the desired data by filtering the records, selecting only those meeting the query criteria. Finally, the records appear on the screen in a datasheet.

These selected records are known as a *dynaset* - a *dynamic set* of data that can change according to the raw data in the original tables. After running a query, the user can use the resulting dynaset in a form, which displays the data in a specified format. In this way, one can limit user access to only the data that meets the criteria in the dynaset. [Ref. 8]

c. *Forms*

Forms help users get information into a database table in a quick, easy, and accurate manner. Data entry and display forms provide a more structured view of the data than does a datasheet. From this structured view, it is possible to view, add, change, or delete database records. Entering data through the data entry forms is the most common way to get the data into the database table.

Data entry forms can be used to restrict access to certain fields within the table. One can also use these forms to check the validity of the data before one accepts it into the database table.

Display-only screens and forms are solely for inquiry purposes. These forms allow for the selective display of certain fields within a given table. Displaying some fields and not others means that the database administrator can limit a user's access to sensitive data while allowing inquiry into other fields. [Ref. 8]

d. Reports

Reports present the data in printed format. It is possible to create several different types of reports within a database management system. This is accomplished by incorporating a query into the report design. The query creates a dynaset consisting of the records that satisfy the conditions.

Reports can combine multiple tables to present complex relationships among different sets of data. An example is printing an invoice. You access the customer table to obtain the customer's name and address and other pertinent data and the sales table to print the individual line item information for the products ordered. You can then have Access calculate the totals and print them in a specific format on the form. [Ref. 8]

e. Macros

A macro is a set of one or more actions that each performs a particular operation, such as opening a form or printing a report. Macros can help the user to automate common tasks. For example, one can run a macro that prints a report when a user clicks a command button.

A macro can be one macro composed of a sequence of actions, or it can be a macro group. You can also use a conditional expression to determine whether in some cases an action will be carried out when a macro runs.

f. Modules

A module is a collection of Visual Basic for Applications declarations and procedures that are stored together as a unit. There are two basic types of modules: class modules and standard modules.

Form and report modules are class modules that are associated with a particular form or report. Form and report modules often contain event procedures that run in response to an event on the form or report. One can use event procedures to control the behavior of forms and reports, and their response to user actions such as clicking the mouse on a command button.

Standard modules contain general procedures that aren't associated with any other object and frequently used procedures that can be run from anywhere within the database.

III. SEMANTIC DATA MODEL

A. INTRODUCTION

The Semantic Database Model (SDM) is a high-level, semantics-based database model that will enable the database designer to naturally and directly incorporate the semantics of a database into its schema. This database model is designed to capture more of the meaning of an application environment than is possible with other database models. An SDM specification describes a database in terms of the kinds of entities that exist in the application environment, the classifications and groupings of those entities, and the structural interconnections among them. SDM provides a collection of high-level modeling primitives to capture the semantics of an application environment. By accommodating derived information in a database structural specification, SDM allows the same information to be viewed in several ways; this makes it possible to directly accommodate the variety of needs and processing requirements typically present in database applications. [Ref. 9]

SDM is designed to enhance the effectiveness and usability of database systems. SDM database description can serve as a formal specification and documentation tool for a database; it can provide a basis for supporting a variety of powerful user interface facilities; and, it can serve as a conceptual database model in the database design process.

SDM has been developed to satisfy a number of criteria that are not met by contemporary database models, but which are essential in an effective database description and design. [Ref. 9]

(1) The constructs of the database model should provide for the explicit specification of a large portion of the *meaning* of a database. Other data models employ overly simple data structures to model an application environment. In so doing, they inevitably lose information about the database; they provide for the expression of only a limited range of a designer's knowledge of the application environment. However, it is essential that the database model provide a rich set of features to allow the direct modeling of application environment semantics.

(2) A database model must support a *relativist* view of the meaning of a database, and allow the structure of a database to support alternative ways of looking at the same information. In order to accommodate multiple views of the same data and to enable the evolution of new perspectives on the data, a database model must support schemata that are flexible, logically redundant, and integrated. *Flexibility* is essential in order to allow for multiple and coequal views of the data. In a *logically redundant* database schema, the values of some database components can be algorithmically derived from others. Finally, an *integrated* schema explicitly describes the relationships and similarities between multiple ways of viewing the same information.

(3) A database model must support the definition of schemata that are based on *abstract entities*. Specifically, this means that a database model must facilitate the description of relevant entities in the application environment, collections of such entities, relationships (associations) among entities, and structural interconnections among the collections. Moreover, the entities themselves must be distinguished from

their syntactic identifiers (*names*); the user-level view of a database should be based on actual entities rather than on artificial entity names.

B. SEMANTIC OBJECTS

A semantic object can be defined as a representation of some identifiable thing in the real world. More formally, a semantic object is a named collection of attributes that sufficiently describes a distinct identity. Semantic objects are grouped into *classes*. An object class has a name that distinguishes it from other classes and that corresponds to the names of the things it represents. [Ref. 3] Thus, a database that supports users who work with student records, has an object class called STUDENT. A particular semantic object is an *instance* of the class. Thus, 'Yuksel Can' is an instance of the STUDENT class.

Like entities, a semantic object has a collection of *attributes*. Each attribute represents a characteristic of the identity being represented. For instance, the STUDENT object could have attributes like Name; HomeAddress; CampusAddress; DateOfBirth; and Major.

Objects represent distinct identities; that is, they are something that users recognize as independent and separate and that users want to track and report. These identities are the nouns about which the information is to be produced. The identities that the objects represent may or may not have a physical existence. For example, STUDENT object represents identities that physically exist, but COURSE does not.

1. Attributes

Semantic objects have attributes that define their characteristics. There are three types of attributes: Simple attributes, Group attributes, and Semantic object attributes.

a. *Simple Attributes*

Simple attributes have a single value. Examples are DateOfBirth, InvoiceNumber, and CourseName.

b. *Group Attributes*

Group attributes are composites of other attributes. One example is Address, which contains the attributes {Street, City, State, Zip}; another example is FullName, which contains the attributes {FirstName, MiddleInitial, LastName}.

c. *Semantic Object Attributes*

Semantic object attributes are attributes that establish a relationship between one semantic object and another.

Attributes may be either single-valued or multi-valued. A *single-valued* attribute is an attribute whose maximum cardinality is 1. A *multi-valued* attribute is one whose maximum cardinality is greater than 1.

Semantic objects are shown in *semantic object diagrams*, which are portrait-oriented rectangles, in which the name of the object appears at the top and the attributes are written in order after the object name. Figure 3.1 shows an example of a semantic object diagram.

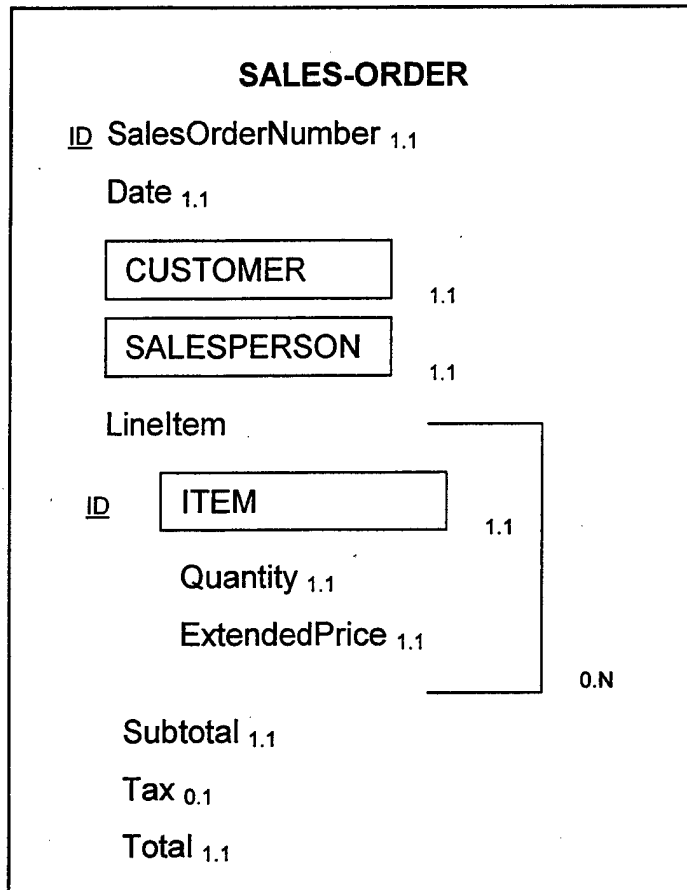


Figure 3.1 Semantic Object Diagram

The SALES-ORDER object contains an example of each of the three types of attributes: SalesOrderNumber, Date, Subtotal, Tax, and Total are all simple attributes, each of which represents a single data element. LinItem is a group attribute containing the simple attributes Quantity and ExtendedPrice, and the semantic object attribute: ITEM. CUSTOMER, SALESPERSON, and ITEM each are semantic object attributes, which means that these objects are related to and logically contained in SALES-ORDER.

2. Attribute Cardinality

Each attribute in a semantic object has both a minimum cardinality and a maximum cardinality. The minimum cardinality indicates the number of instances of the attribute that must exist in order for the object to be valid. Usually this number is either "0" or "1". If it is 0, the attribute is not required to have a value. If it is 1, the attribute must have a value. The maximum cardinality indicates the maximum number of instances of the attribute that the object may have. It is usually either 1 or N. If it is 1, the attribute can have no more than one instance; if it is N, the attribute can have many values.

[Ref. 3]

Cardinalities are shown as subscripts of attributes in the format **n.m**, where n is the minimum cardinality and m is the maximum. In Figure 3.1, the minimum cardinality of SalesOrderNumber is 1 and the maximum is also 1, which means that exactly one value of SalesOrderNumber is required. The cardinality of 0.1 in Tax means that a SALES-ORDER may have either zero or one Tax.

The cardinalities of group attributes and the attributes inside groups can be interpreted as follows: The cardinality of LineItem group attribute is 0.N, meaning that a SALES-ORDER may have zero or more line items. However, each attribute within this group has a cardinality of 1.1, meaning that these attributes are required. One might wonder how a group could be optional if the attributes in that group are required. The answer is that the cardinalities operate only between the attribute and the container of that attribute. Thus, a LineItem group need not appear in a SALES-ORDER, but if it does, then it must have a value for Item, Quantity, and ExtendedPrice attributes.

3. Paired Attributes

The semantic object model has no one-way relationships. If an object contains another object, the second object will contain the first one. For example, if SALES-ORDER contains the object attribute SALESPERSON, then SALESPERSON will contain the matching object attribute SALES-ORDER. The object attributes always occur as a pair, because if Object A has a relationship with Object B, then Object B will have a relationship with Object A. [Ref. 3]

4. Object Identifiers

An *object identifier* is one or more object attributes that the users employ to identify object instances. In SALES-ORDER semantic object, for example, the object identifier is SalesOrderNumber, an attribute that uniquely identifies each Sales-Order instance. A *group identifier* is an identifier that has more than one attribute. Examples are {FirstName, LastName}, {Firstname, PhoneNumber}, and {State, LicenseNumber}.

Object identifiers may or may not be unique, depending on the type of the data. For example, SalesOrderNumber is a unique identifier for SALES-ORDER, but StudentName is not a unique identifier for STUDENT, because there may be two students named 'Mary Smith'.

In semantic object diagrams, object identifiers are denoted by the letters *ID* in front of the attribute. If the identifier is unique, these letters will be underlined. In Figure 3.1, for example, the attribute SalesOrderNumber is a unique identifier of SALES-ORDER object class.

5. Attribute Domains

The *domain* of an attribute is a description of an attribute's possible values. The characteristics of a domain depend on the type of the attribute. The domain of a simple attribute consists of both a physical and a semantic description. The *physical description* indicates the type of data (for example, numeric versus string), the length of the data, and other restrictions or constraints (such as the value must not exceed 100). The *semantic description* indicates the function or purpose of the attribute-it distinguishes this attribute from other attributes that might have the same physical description. [Ref. 3]

The domain of a group attribute also has a physical and a semantic description. The physical description is a list of all of the attributes in the group and the order of those attributes. The semantic description is the function or purpose of the group.

The domain of an object attribute is the set of object instances of that type. In Figure 3.1, for example, the domain of the CUSTOMER object attribute is the set of all CUSTOMER object instances in the database.

C. TYPES OF SEMANTIC OBJECTS

This section describes and illustrates seven types of semantic objects. [Ref. 3]

1. Simple Objects

A simple object is a semantic object that contains only single-valued, non-object attributes. [Ref. 3] Figure 3.2 shows a simple object, EQUIPMENT, that models Equipment Tag. None of the attributes of this object is multi-valued, and none is an object attribute.

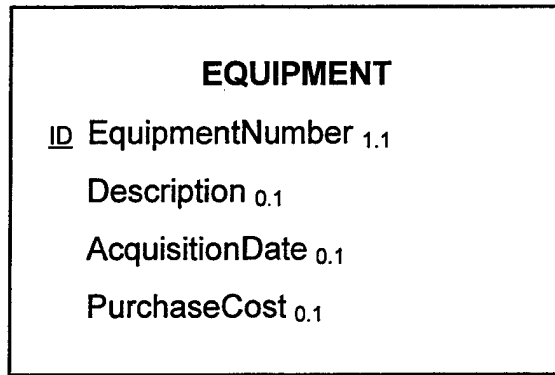


Figure 3.2: EQUIPMENT Simple Object

2. Composite Objects

A composite object is a semantic object that contains one or more multi-valued, non-object attributes. [Ref. 3] The HOTEL-BILL object, shown in Figure 3.3, is a composite object that contains a multi-valued group attribute, LineItem.

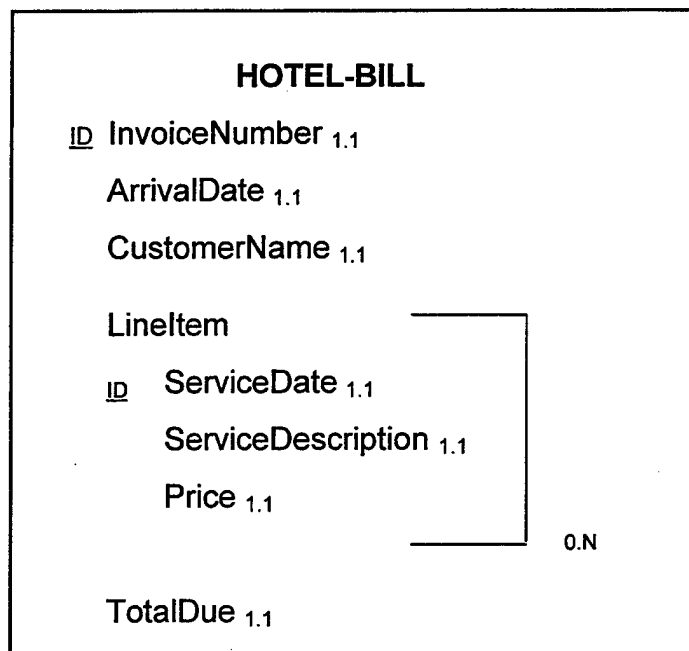


Figure 3.3: HOTEL-BILL Composite Object

3. Compound Objects

A compound object is a semantic object that contains at least one object attribute. The relationship between two compound objects can be one to one (1:1), one to many (1:N), or many to many (M:N). [Ref. 3] An illustration of compound objects that have a M:N relationship appears in Figure 3.4. From these object diagrams, we can deduce that one book can be written by many authors and that one author can write many books, because BOOK object contains many values of AUTHOR, and AUTHOR contains many values of BOOK. Hence the relationship from BOOK to AUTHOR is many to many or N:M. Furthermore, a BOOK must have an AUTHOR, and an AUTHOR (to be an author) must have written at least one BOOK. Therefore, both of these objects have a minimum cardinality of one.

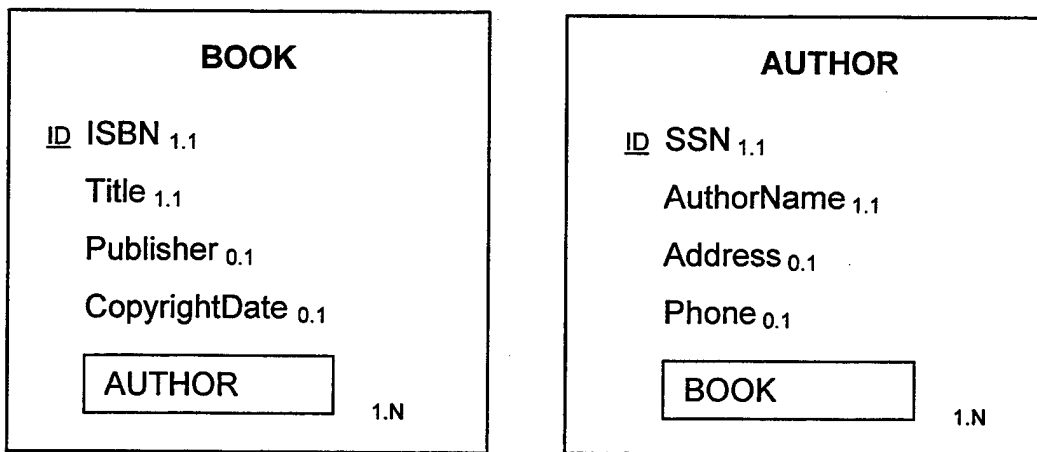


Figure 3.4: BOOK and AUTHOR Compound Objects

4. Hybrid Objects

Hybrid objects are combinations of compound and composite objects. In particular, a hybrid object is a semantic object with at least one multi-valued group

attribute that includes a semantic object attribute. Figure 3.5 is an object diagram that models a hybrid object. SALES-ORDER contains a multi-valued group attribute, LineItem, with both the object attribute ITEM and the non-object attributes Quantity and ExtendedPrice. This means that Quantity and ExtendedPrice are paired with ITEM in the context of SALES-ORDER.

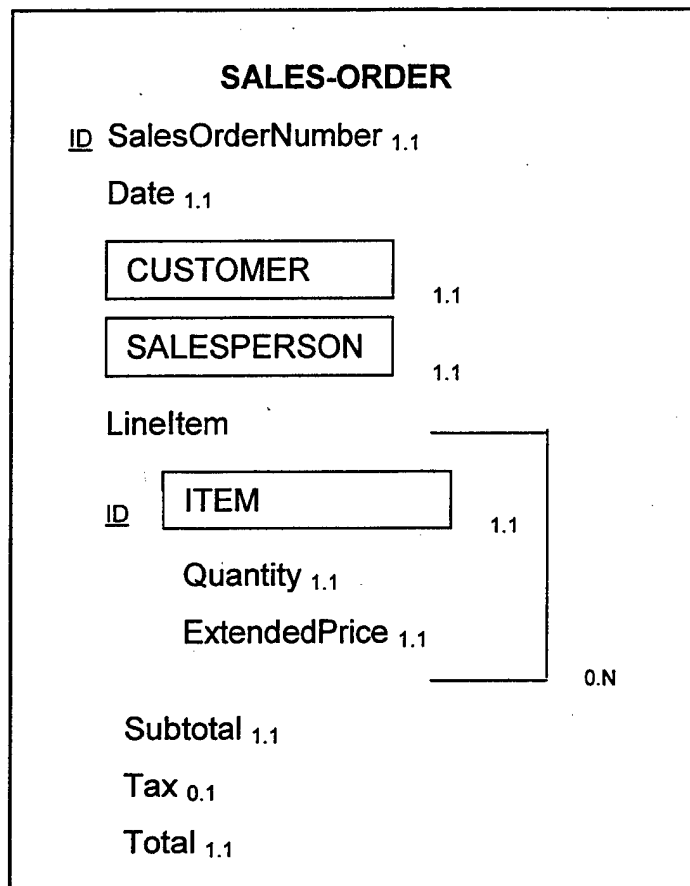


Figure 3.5: SALES-ORDER Hybrid Object

5. Association Objects

An association object is an object that relates two or more objects and stores data that are peculiar to that relationship. [Ref. 3] In Figure 3.6, the object FLIGHT is an

association object that associates the two objects AIRPLANE and PILOT and stores data about their association. FLIGHT contains one each of AIRPLANE and PILOT; but both AIRPLANE and PILOT contain multiple values of FLIGHT. This particular pattern of associating two or more objects with data about the association occurs frequently in applications that involve the assignment of two or more things.

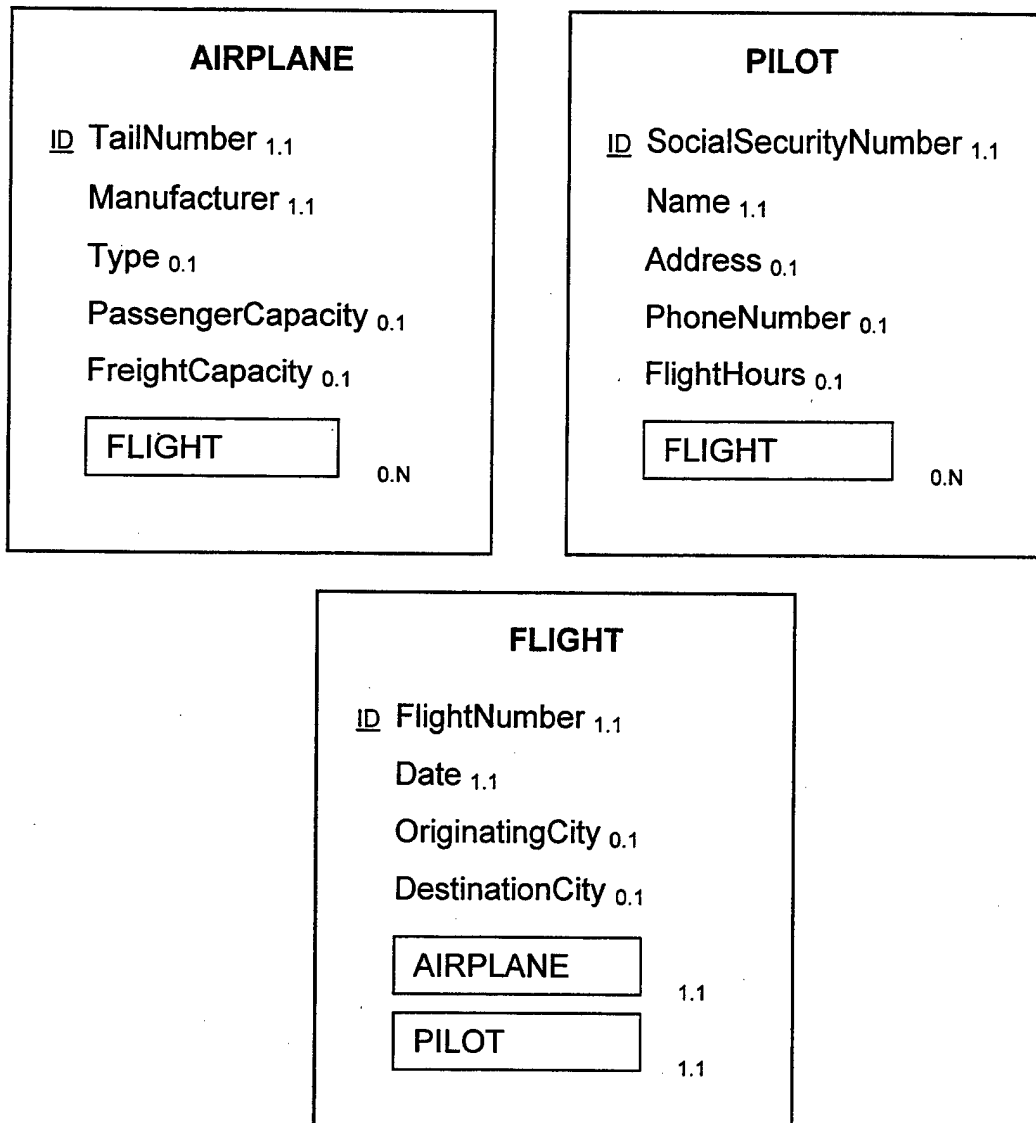


Figure 3.6: FLIGHT, AIRPLANE, and PILOT Semantic Objects

6. Parent/Subtype Objects

Parent/Subtype Objects model generalization, inheritance and is-a relationships. [Ref. 3] An example is shown in Figure 3.7, in which the EMPLOYEE object contains a subtype object, MANAGER. EMPLOYEE object has the common attributes of all employees, while MANAGER object having manager-oriented attributes. In this example, the EMPLOYEE object is called a *parent object or supertype object*, and the MANAGER object is called a *subtype object*.

The first attribute of a subtype object is the parent object attribute and is denoted by the subscript P. Parent attributes are always required in subtype objects. The identifiers of the subtype are the same as the identifiers of the parent.

Subtype attributes are shown with the subscript 0.ST or 1.ST. The first digit is the minimum cardinality of the subtype. If it is 0, the subtype is optional, and if it is 1, the subtype is required. The ST indicates that the attribute is a subtype, or IS-A attribute.

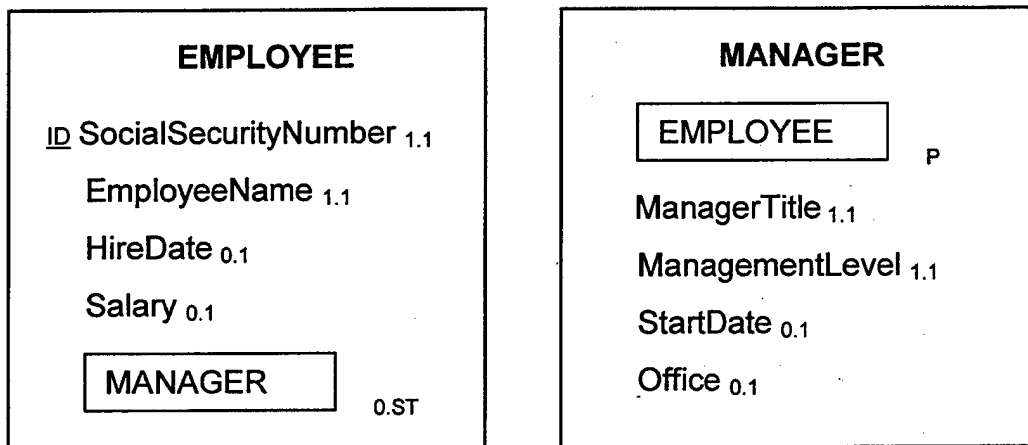


Figure 3.7: EMPLOYEE Supertype and MANAGER Subtype Objects

Parent/subtype objects have an important characteristic, called *inheritance*. A subtype acquires, or *inherits* all of the attributes of its parent, and therefore a MANAGER inherits all of the attributes of an EMPLOYEE.

A semantic object may contain more than one subtype attribute. In this case, the parent object contains each subtype as an attribute. If the subtypes exclude one another, they are placed into a subtype group, and the group is assigned a subscript of the format X.Y.Z. X is the minimum cardinality and is 0 or 1, depending on whether or not the subtype group is required. Y and Z are counts of the number of attributes in the group that are allowed to have a value. Y is the minimum number required, and Z is the maximum number allowed. [Ref. 3]

Figure 3.8 shows three types of CLIENT as a subtype group. The subscript of the group, 0.1.1, means that the subtype is not required, but if it exists, a minimum of one and a maximum of one (or exactly one) of the subtypes in the group must exist.

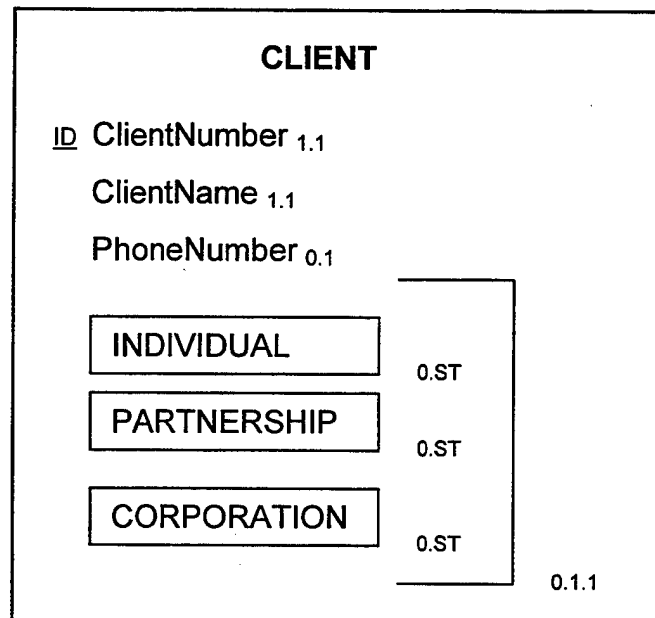


Figure 3.8: Exclusive Subtypes

Each of the subtypes has the subscript 0.ST, meaning that they all are optional. This notation is robust enough to allow for situations in which three out of five or seven out of ten of a list of subtypes must be required.

7. Archetype/Version Objects

An archetype object is a semantic object, which produces other semantic objects that represent versions, releases, or editions of the archetype. [Ref. 3] For example, in Figure 3.9, the archetype object TEXTBOOK produces the version object EDITION. According to this model, the attributes Title, Author, and Publisher belong to the object TEXTBOOK, and the attributes EditionNumber, Publication Date, and NumberOfPages belong to the EDITION of the TEXTBOOK.

The ID group in EDITION has two portions, TEXTBOOK and EditionNumber; this is the typical pattern for an ID of a version object. One part of the ID contains the archetype object, and the second part is a simple attribute that identifies the version within the archetype.

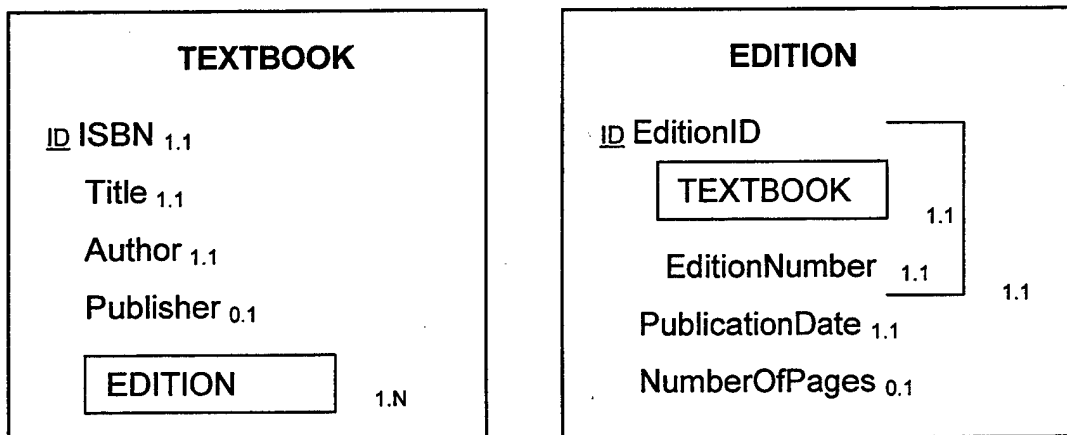


Figure 3.9: TEXTBOOK Archetype and EDITION Version Objects

D. TRANSFORMATION OF SEMANTIC OBJECTS INTO RELATIONS

This section discusses the transformation of semantic object models into relational database designs by describing the transformation of each semantic object type.

1. Transformation of Simple Objects

Recall that a simple object has no multi-valued attributes and no object attributes. Consequently, a simple object can be represented by a single relation in the database. Figure 3.10 (a) is an example of a simple object, EQUIPMENT, which can be represented by a single relation, as shown in Figure 3.10 (b). Each attribute of the object is defined as an attribute of the relation, and the identifying attribute, EquipmentNumber, becomes the primary key of the relation, denoted by underlining and making EquipmentNumber boldface.

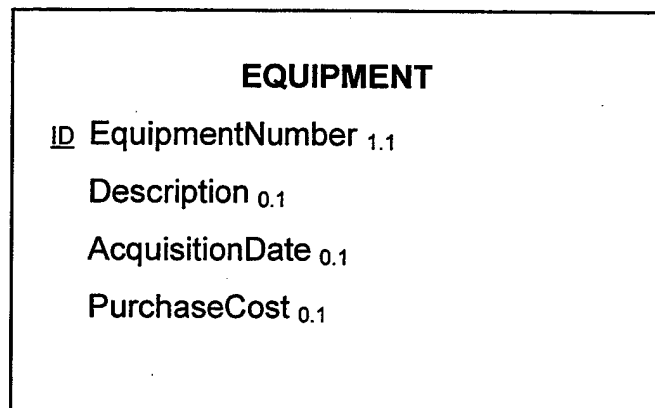


Figure 3.10 (a): EQUIPMENT Simple Object

EQUIPMENT (**EquipmentNumber**, Description, AcquisitionDate, PurchaseCost)

Figure 3.10 (b): EQUIPMET Relation

In general, simple objects are transformed into relations by creating a relation for each simple object.

2. Transformation of Composite Objects

A composite object is an object that has one or more multi-valued simple or group attributes but no object attributes. Figure 3.11 (a) shows an example composite object, HOTEL-BILL. To represent this object, one relation is created for the base object, HOTEL-BILL, and an additional relation is created for the repeating group attribute, LineItem. This relational design is shown in Figure 3.11 (b). In the key of LINEITEM, InvoiceNumber is underlined because it is part of the key, and it is italicized because it is also a foreign key.

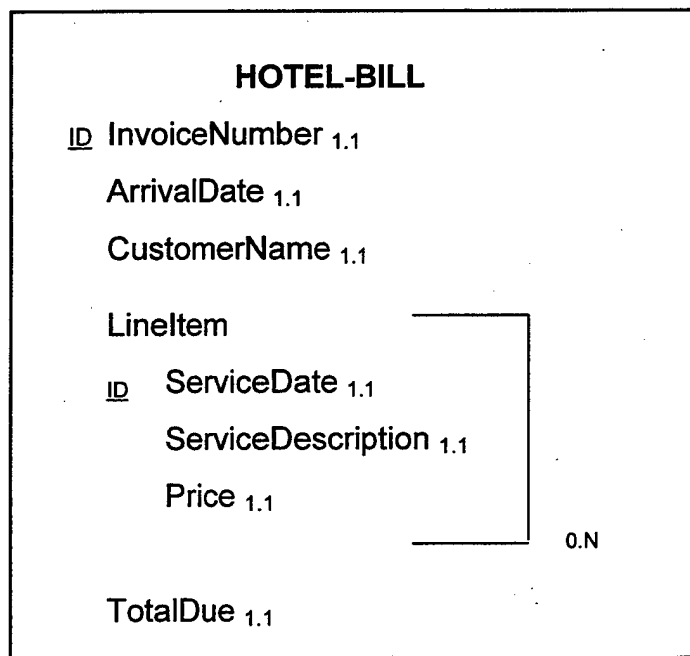


Figure 3.11 (a): HOTEL-BILL Composite Object

HOTEL-BILL (InvoiceNumber, ArrivalDate, CustomerName, TotalDue)

LINEITEM (InvoiceNumber, ServiceDate, ServiceDescription, Price)

Figure 3.11 (b): HOTEL-BILL and LINEITEM Relations

In general, composite objects are transformed by defining one relation for the object itself and another relation for each multi-valued attribute. The key of the tables constructed for the multi-valued attributes is the composite of the identifier of the object plus the identifier of the group.

3. Transformation of Compound Objects

A compound object, OBJECT1, can contain one or many instances of a second object, OBJECT2, and OBJECT2 can contain one or many instances of the first object, OBJECT1. This leads to three types of relationships between compound objects: one to one (1:1), one to many (1:N), and many to many (M:N).

a. Representing 1:1 Compound Objects

Figure 3.12 (a) shows the object diagrams for one to one compound objects MEMBER and LOCKER. To represent these objects with relations, a relation for each object is defined, and a key of either relation is placed in the other relation. That is, one can place the key of MEMBER in LOCKER or the key of LOCKER in MEMBER. Figure 3.12 (b) shows the placement of the key of LOCKER in MEMBER. Note that LockerNumber is underlined in LOCKER because it is the primary key of LOCKER and is italicized in MEMBER because it is a foreign key in MEMBER.

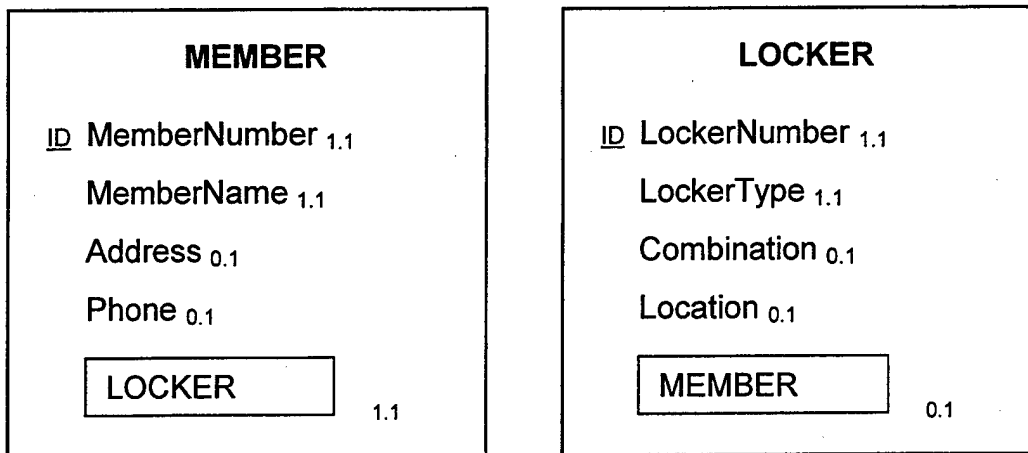


Figure 3.12 (a): One-to-One Compound Objects

MEMBER (MemberNumber, MemberName, Address, Phone, LockerNumber)

LOCKER (LockerNumber, LockerType, Combination, Location)

Figure 3.12 (b): MEMBER and LOCKER Relations

In general, for a 1:1 relationship between two compound objects, we define one relation for each object, and we place the key of either relation as a foreign key in the other relation.

b. Representing 1:N Compound Objects

Figure 3.13 (a) shows an example of a 1:N object relationship between EQUIPMENT and REPAIR. An item of EQUIPMENT can have many REPAIRs, but a REPAIR can be related to only one item of EQUIPMENT. The objects in Figure 3.13 (a) are represented by the relations in Figure 3.13 (b). Observe that the key of the parent (the object on the one side of the relationship) is placed in the child (the object on the many side of the relationship).

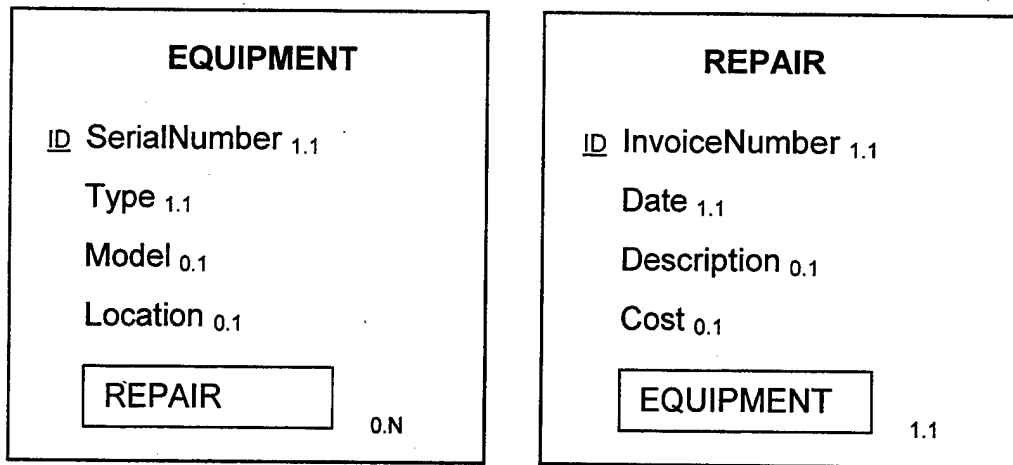


Figure 3.13 (a): One-to-Many Compound Objects

EQUIPMENT (SerialNumber, Type, Model, Location)

REPAIR (InvoiceNumber, Date, Description, Cost, *SerialNumber*)

Figure 3.13 (b): EQUIPMENT and REPAIR Relations

One to many compound objects are transformed into relations by representing each object with a relation and placing the key of the parent (the object on the one side of the relationship) in the child (the object on the many side of the relationship).

c. Representing M:N Compound Objects

Figure 3.14 (a) shows the M:N relationship between BOOK and AUTHOR. Figure 3.14 (b) depicts the three relations that represent these objects: BOOK, AUTHOR, and BOOK-AUTHOR-INTERSECTION. The attributes of BOOK-AUTHOR- INTERSECTION relation are underlined and in italics, because they both are local and foreign keys.

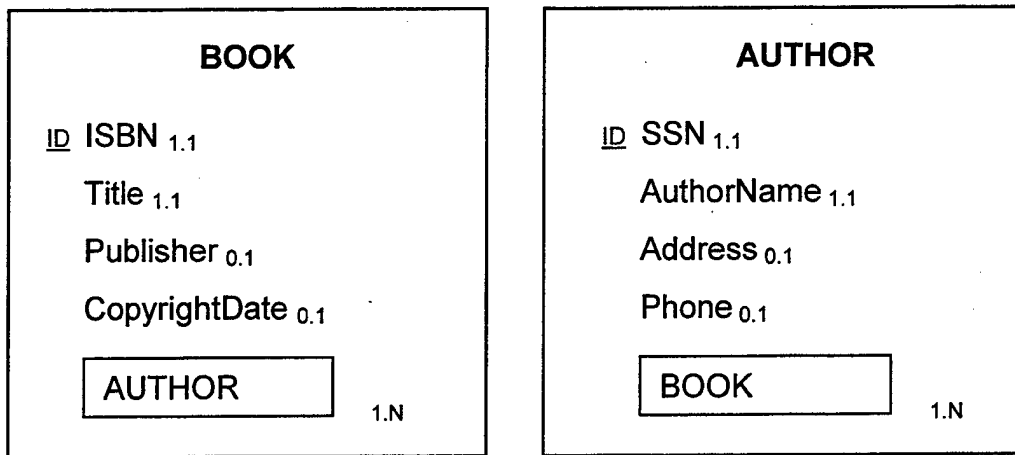


Figure 3.14 (a): BOOK and AUTHOR Compound Objects

BOOK (ISBN, Title, Publisher, CopyrightDate)

AUTHOR (SSN, AuthorName, Address, Phone)

BOOK-AUTHOR-INTERSECTION (ISBN, SSN)

Figure 3.14 (b): BOOK, AUTHOR, and BOOK-AUTHOR-INTERSECTION Relations

In general, for two objects that have an M:N relationship, it is necessary to define three relations; one for each of the objects and a third intersection relation. The intersection relation represents the relationship of the two objects and consists of the keys of both of its parents.

4. Transformation of Hybrid Objects

Hybrid objects can be transformed into relational designs using a combination of the techniques for composite and compound objects. Figure 3.15 (a) shows SALES-ORDER hybrid object, and related objects.

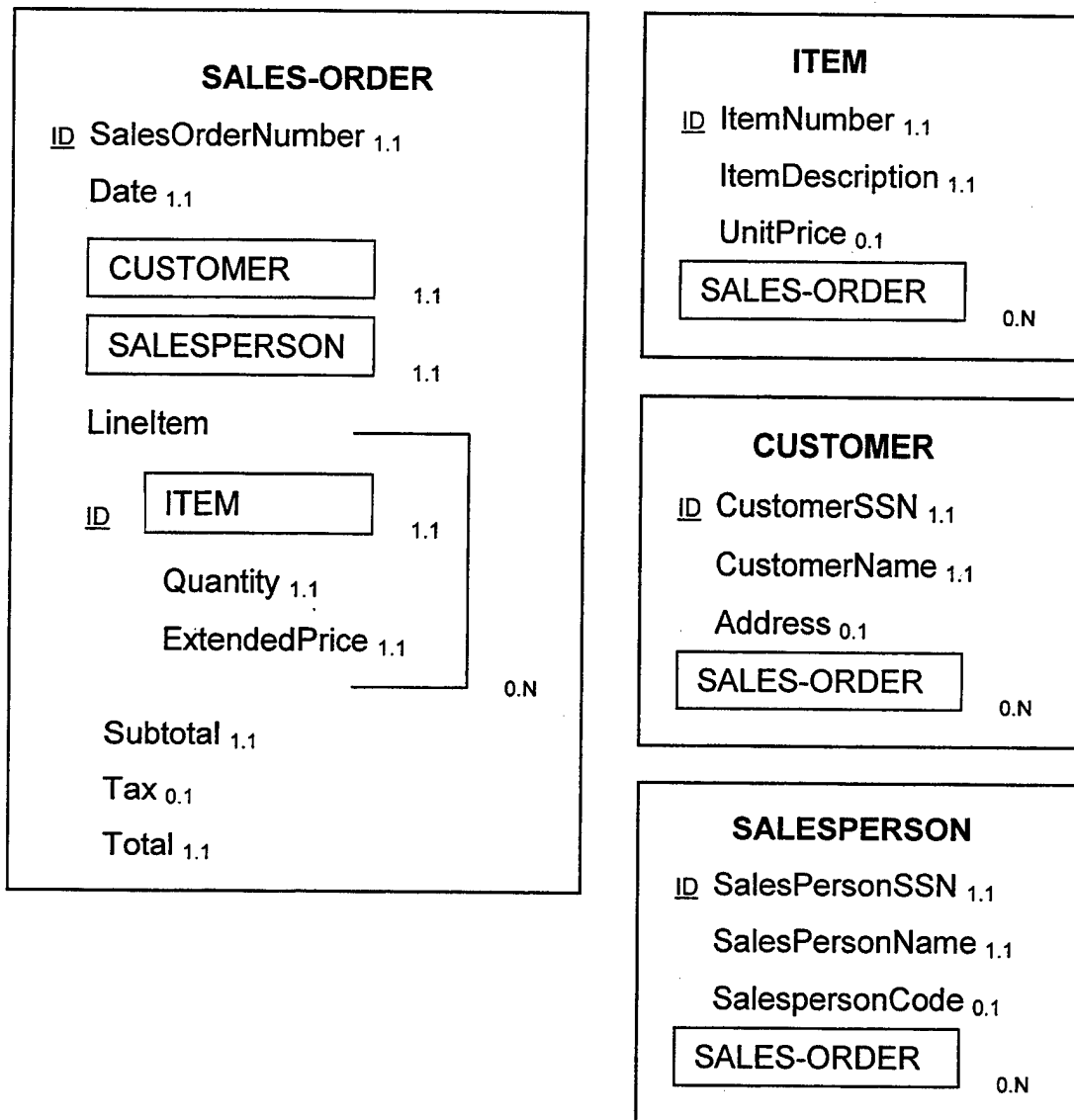


Figure 3.15 (a): SALES-ORDER Hybrid Object and ITEM, CUSTOMER, and SALESPERSON Compound Objects

SALES-ORDER (SalesOrderNumber, Date, Subtotal, Tax, Total,
CustomerSSN, SalesPersonSSN)

CUSTOMER (CustomerSSN, CustomerName, Address)

SALESPERSON (SalesPersonSSN, SalesPersonName,
SalespersonCode)

ITEM (ItemNumber, ItemDescription, UnitPrice)

LINEITEM (SalesOrderNumber, ItemNumber, Quantity, ExtendedPrice)

**Figure 3.15 (b): SALES-ORDER, ITEM, CUSTOMER, SALESPERSON, and
LINEITEM Relations**

To represent this object by means of relations, we establish one relation for the object itself and another relation for each of the contained objects CUSTOMER and SALESPERSON. Then, as with a composite object, it is needed to define a relation for the multi-valued group, which is LineItem. Since this group contains another object, ITEM, a relation is created for ITEM. All of the one to many relationships are represented by placing the key of the parent relation in the child relation, as shown in Figure 3.15 (b).

5. Transformation of Association Objects

An association object is an object that associates two other objects. It is a special case of compound objects that most often occurs in assignment situations. Figure 3.16 (a) shows a FLIGHT object that associates an AIRPLANE with a PILOT. To represent association objects, we define a relationship for each of the three objects, and then we represent the relationships among the objects using one of the strategies used with compound objects. In Figure 3.16 (b), one relation is defined for AIRPLANE, one for PILOT and one for FLIGHT. The relationships between FLIGHT and AIRPLANE and between FLIGHT and PILOT are 1:N, so we place the key of the AIRPLANE and the key of the PILOT in the FLIGHT.

In general, when transforming association object structures into relations, one relation is defined for each of the objects participating in the relationship. The key of each of the parent relations appears as foreign key attributes in the relation representing the association object. If the association object has no unique identifying attribute, the combination of the attributes of the parent relations will be used to create a unique identifier.

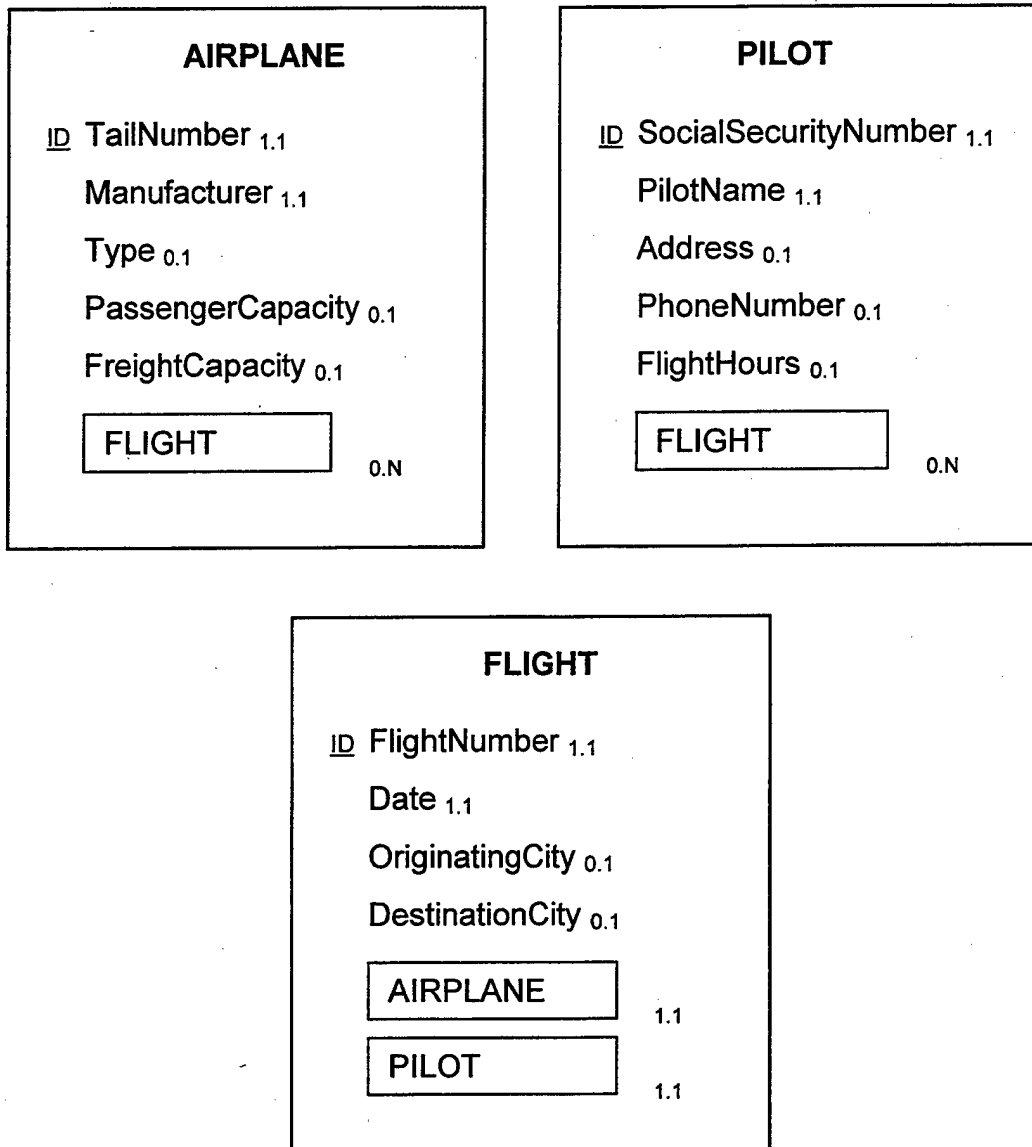


Figure 3.16 (a): FLIGHT Association Object and AIRPLANE and PILOT Compound Objects

AIRPLANE (TailNumber, Manufacturer, Type, PassengerCapacity,
FreightCapacity)

PILOT (SocialSecurityNumber, PilotName, Address, PhoneNumber,
FlightHours)

FLIGHT (FlightNumber, Date, OriginatingCity, DestinationCity,
TailNumber, SocialSecurityNumber)

Figure 3.16 (b): AIRPLANE, PILOT, and FLIGHT Relations

6. Transformation of Parent/Subtype Objects

To transform parent/subtype objects into relations, it is necessary to define a relation for the parent object and one for each of the subtype objects. The key of each of these relations is the key of the parent object.

Figure 3.17 (a) shows a parent object, EMPLOYEE, and a subtype object, MANAGER. Figure 3.17 (b) shows the relational representation of these two objects. Each object is represented by a table, and the primary key of all of the tables is the same.

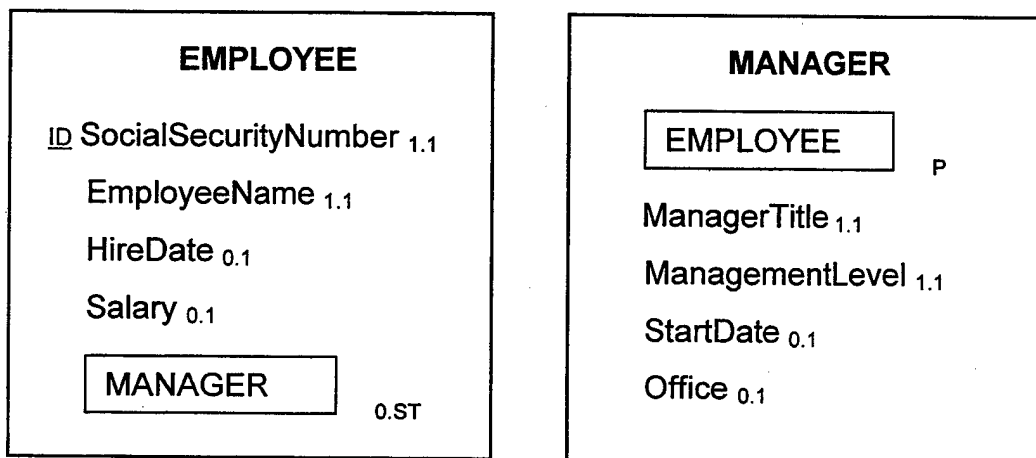


Figure 3.17 (a): EMPLOYEE Supertype and MANAGER Subtype Objects

EMPLOYEE (SocialSecurityNumber, EmployeeName, HireDate, Salary)

MANAGER (SocialSecurityNumber, ManagerTitle, ManagementLevel,
StartDate, Office)

Figure 3.17 (b): EMPLOYEE and MANAGER Relations

7. Transformation of Archetype/Version Objects

Archetype/version objects are compound objects that model various iterations, releases, or instances of a basic object. The objects in Figure 3.18 (a) model textbooks for which there are various editions. The relational representation of TEXTBOOK and EDITION is shown in Figure 3.18 (b). One relation is created for TEXTBOOK, and another is created for EDITION. The primary key of EDITION is the combination of the key of TEXTBOOK and the local key (EditionNumber) of EDITION.

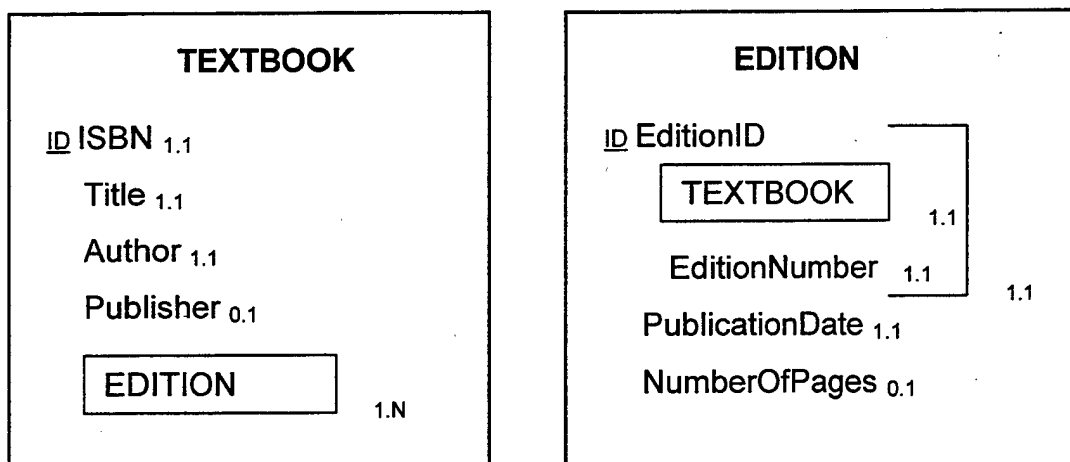


Figure 3.18 (a): TEXTBOOK Archetype and EDITION Version Objects

TEXTBOOK (ISBN, Title, Author, Publisher)

EDITION (ISBN, EditionNumber, PublicationDate, NumberOfPages)

Figure 3.18 (b): TEXTBOOK and EDITION Relations

IV. JAVA AND JAVA DATABASE CONNECTIVITY (JDBC)

The user will interact with the back-end database of the POET database system by using the application program which is developed in Java programming language. The application program written in Java will communicate with the database by using Java Database Connectivity (JDBC) application program interface. This chapter will describe Java and JDBC application programming interface (API) and how they can be used to provide this type of interaction. It will summarize the attributes of Java programming language and outline the JDBC API, classes, methods, and how they can be used by applications to directly access a RDBMS.

A. JAVA

Java is a powerful and fully object-oriented programming language that makes it possible to program for the Internet by creating applets, programs that can be embedded in a web page. Instead of web pages with text and static graphics, Java applets can make use of audio, animation, interactivity and video imaging.

But Java is more than a programming language for writing applets. It is being used extensively for writing standalone applications as well. It is becoming so popular that many people believe it will become the standard language for both general-purpose and Internet programming.

Java is actually a platform consisting of three components: (1) the Java programming language, (2) the Java library of classes and interfaces, and (3) the Java Virtual Machine.

Java programs go through five phases in order to be executed. These are edit, compile, load, bytecode verify, and execute. Figure 4.1 describes the specifics of a Java development environment.

1. Editing

The Java source code is created and saved on disk with the file extension `.java`.

2. Compiling

The Java compiler creates the bytecode for the program and stores it on disk.

3. Loading

The class loader loads the bytecode into memory. The Java interpreter acts as the class loader for Java applications

4. Bytecode verification

Bytecode verifier confirms that all bytecode is valid and does not violate any of Java's security restrictions

5. Execution

Interpreter reads the bytecode and translates it into a machine language that the computer can understand.

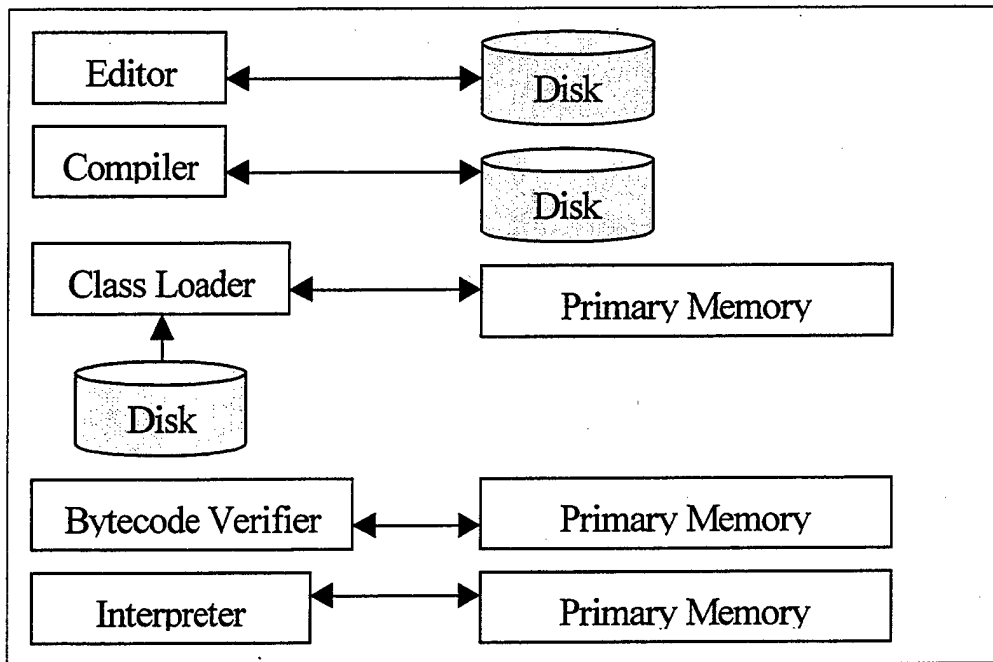


Figure 4.1: Typical Java Environment [Ref: 5]

B. ADVANTAGES OF JAVA

Java builds on the strengths of C++. It has taken the best features of C++ and discarded the more problematic and error-prone parts. To this lean core it has added garbage collection (automatic memory management), multithreading (the capacity for one program to do more than one thing at a time), and security capabilities. The result is that Java is simple, elegant, powerful, and easy to use. [Ref. 10]

The following section will explain twelve features and advantages of Java in detail.

1. Java Is Portable

One of the biggest advantages Java offers is that it is portable. Therefore, an application written in Java is platform independent. Any computer with a Java-based browser can run the applications or applets written in the Java programming language. Developers don't need to modify applets or stand-alone applications code when changing platforms.

The Java Virtual Machine is what gives Java its cross-platform capabilities. Rather than being compiled into a machine language, which is different for each operating system and computer architecture, Java code is compiled into bytecodes.

With other programming languages, the compiler creates platform specific machine language code. The problem is that other computers with different machine instruction sets cannot understand that language. Java code, on the other hand, is compiled into bytecodes rather than a machine language. These bytecodes go to the Java Virtual Machine, which executes them directly or translates them into the language that is understood by the machine running it. [Ref. 10]

In summary, this means that with the JDBC API extending Java, a programmer writing Java code can access all of the major relational databases on any platform that supports the Java Virtual Machine.

2. Java Is Object-Oriented

The Java programming language was designed from the start as an object-oriented programming language. Simply stated, object-oriented design is a technique for programming that focuses on the objects and on the interfaces to that object. Object-

oriented languages make program design focus on *what* to deal with rather than on *how* to do something and enable designers to break up large projects into easily manageable components. Another big benefit is that these components can then be reused. [Ref. 10]

Object-oriented languages use the paradigm of classes. In simplest terms, a class includes both data and the functions to operate on that data. One can create an *instance* of a class, also called an *object*, which will have all the data members and functionality of its class. Because of this, it is possible to think of a class as being like a template, with each object being a specific instance of a particular type of class.

The class paradigm allows one to *encapsulate* data so that specific data values or function implementations cannot be seen by those using the class. Encapsulation makes it possible to make changes in code without breaking other programs, which use that code. If, for example, the implementation of a function is changed, the change is invisible to another programmer who might invoke that function, and it doesn't affect his/her program.

Java includes *inheritance*, or the ability to derive new classes from existing classes. The *derived* class, also called a *subclass*, inherits all the data and functions of the existing class, referred to as the *parent* class. A subclass can add new data members to those inherited from the parent class. As far as methods are concerned, the subclass can reuse the inherited methods, as is, change them, and/or add its own new methods.

3. Java Makes It Easy to Write Correct Code

In addition to being portable and object-oriented, Java facilitates writing correct code. Programmers spend less time writing Java code and a lot less time debugging it.

The following is a list of some of Java's features that make it easier to write correct code:

[Ref. 10]

a. *Garbage collection*

With other programming languages a significant burden on the programmer is the allocation and de-allocation of memory. Memory leaks in a program can cause an application to crash. In Java, if an object is no longer being used, it is automatically removed from memory by the Java garbage collector. Programmers don't have to keep track of what has been allocated and deallocated, but, more importantly, it stops memory leaks. [Ref. 10]

b. *No Pointers*

The No Pointers feature removes a significant source of errors in computer programming. Java utilizes "object references" instead of memory pointers. This eliminates problems concerning pointer arithmetic and "out of bounds" memory access errors.

c. *Strong Typing*

Java enforces strong type checking, therefore many errors are caught at compile time. Dynamic binding is possible and often very useful, but static binding with strict type checking is used when possible. This significantly cuts down on run-time errors.

d. Simplicity

Java keeps it simple by having one way to do something instead of having several alternatives, as in some languages. The syntax for Java is a cleaned-up version of the syntax for C++. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on. Java also stays lean by not including multiple inheritance, which eliminates the errors and ambiguity that arise when you create a subclass that inherits from two or more classes. To replace the capabilities multiple inheritance provides, Java lets you add functionality to a class through the use of *interfaces*. Such capabilities make Java easier to learn and use correctly.

4. Java Includes a Library of Classes and Interfaces

The Java platform includes an extensive class library so that programmers can use already-existing classes as is, create subclasses to modify existing classes, or implement interfaces to augment the capabilities of classes. [Ref. 10]

Both classes and interfaces contain data members (fields) and functions (methods), but there are major differences. In a class, fields may be either variable or constant, and methods are fully implemented. In an interface, fields must be constants, and methods are just prototypes with no implementations. To use an interface, a programmer defines a class, declares that it implements the interface, and then implements all of the methods in that interface as part of the class. In other words, interfaces provide most of the advantages of multiple inheritance without its disadvantages.

5. Java Is Extensible

A big plus for Java is the fact that it can be extended. It was purposely written to be lean with the emphasis on doing what it does very well; instead of trying to do everything from the beginning, it was written so that extending it is easy. Programmers can modify existing classes or write their own new classes, or they can write a whole new package. The JDBC API, the `java.sql` package, is one example of a foundation upon which extensions are being built. [Ref. 10]

6. Java Is Secure

It is important that a programmer not be able to write subversive code for applications or applets. This is especially true with the Internet being used more and more extensively for services such as electronic commerce and electronic distribution of software and multimedia content.

The Java platform builds in security in four ways:

a. The way memory is allocated and laid out

In Java, an object's location in memory is not determined until run time, as opposed to C and C++, where the compiler makes memory layout decisions. As a result, a programmer cannot look at a class definition and figure out how it might be laid out in memory. Also, since Java has no pointers, a programmer cannot forge pointers to memory. [Ref. 10]

b. The way incoming code is checked

The Java Virtual Machine does not trust any incoming code and subjects it to what is called *bytecode verification*. The bytecode verifier, part of the Virtual Machine, checks that (1) the format of incoming code is correct, (2) incoming code doesn't forge pointers, (3) it doesn't violate access restrictions, and (4) it accesses objects as what they are.

c. The way classes are loaded

The Java bytecode loader, another part of the Virtual Machine, checks whether classes loaded during program execution are local or from across a network. Imported classes cannot be substituted for built-in classes, and built-in classes cannot accidentally reference classes brought in over a network.

d. The way access is restricted for untrusted code

The Java security manager allows users to restrict untrusted Java applets so that they cannot access the local network, local files, and other resources.

7. Java Is Multithreaded

Multithreading is simply the ability of a program to do more than one thing at a time. The benefits of multithreading are better interactive responsiveness and real-time behavior [Ref. 11]. For example, an application could be faxing a document at the same time it is printing another document.

Or a program could process new inventory figures while it maintains a feed of current prices. Threads in Java also have the capacity to take advantage of multiprocessor systems.

8. Java Performs Well

Java's many advantages, such as having built-in security and being interpreted as well as compiled, do have a cost attached to them. However, various optimizations have been built in, and the bytecode interpreter can run fast because it does not have to do any checking. For situations that require unusually high performance, bytecodes can be translated on the fly, generating the final machine code for the particular CPU on which the application is running, at run time. Java offers good performance with the advantages of high-level languages but without the disadvantages of C and C++. [Ref. 10]

9. Java scales Well

The Java platform is designed to scale well, from portable consumer electronic devices (PDAs) to powerful desktop and server machines. The Java Virtual Machine takes a small footprint, and Java bytecode is optimized to be small and compact. As a result, Java accommodates the need for low storage and for low bandwidth transmission over the nternet. This makes Java ideal for low-cost network computers whose sole purpose is to access the Internet. [Ref. 10]

10. Java Is Distributed

Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system. The networking capabilities of Java is both strong and easy to use. Java even makes Common Gateway Interface (CGI) scripting easier, and an elegant mechanism called *servlets*, makes server-side processing in Java extremely efficient. The Remote Method Invocation (RMI) mechanism enables communication between distributed objects. [Ref. 11]

11. Java Is Robust

Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (run-time) checking, and eliminating situations that are error-prone. The Java compiler detects many problems that, in other languages, would only show up at run time. [Ref. 11]

12. Java Is Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It is designed to adapt to an evolving environment. Class libraries can freely add new methods and instance variables without any effect on their clients. This is an important feature in those situations where code needs to be added to a running program. [Ref. 11]

C. JDBC

Java Database Connectivity (JDBC) is an Application Program Interface (API) developed by Sun Microsystems, that allows a Java program to communicate with a database server using Structured Query Language (SQL) commands. It provides Java programs the ability to communicate with relational database management systems similar to Microsoft's Open Database Connectivity (ODBC) API [Ref. 4].

Java, being robust, secure, easy to use, easy to understand, and automatically downloadable on a network, is an excellent language basis for database applications. What is needed is a way for Java applications to talk to a variety of different databases. JDBC is the mechanism for doing this. Using JDBC, it is easy to send SQL statements to virtually any relational database. [Ref. 10]

There are four main steps in accessing a database using JDBC:

- Load the driver
- Establish a connection with the database
- Send SQL statements
- Process the results

The following section describes each step in detail:

1. Loading the driver

Loading the driver is accomplished by asking for an instance of the driver explicitly, as in the following line:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```


There is no need to create an instance of a driver and register it with the Driver Manager class because calling `Class.forName` will do that automatically.

2. Establishing a connection with the database

A connection is established by using the appropriate driver to connect to the DBMS. The corresponding sub-protocol identifier is appended after the keyword `jdbc:`. The following code explains how this is accomplished:

```
String user = "yuksel";  
String password = "poet";  
String url = "jdbc:odbc:POETDB";  
Connection con = DriverManager.getConnection  
                (url, user, password);
```

3. Sending SQL statements

In order to execute a SQL statement on a relational database using JDBC, a statement object must be created. The SQL statement to be executed is supplied as an argument to the proper method of the statement object. The type of method differs according to the query being executed. The following code shows how to create a statement object and execute a simple UPDATE query:

```
String query = "UPDATE Personnel " +
               "SET PhoneNumber = '(831)372-4408' " +
               "WHERE LastName = 'Can' ";
Statement stmt = con.createStatement();
stmt.executeUpdate(query);
```

4. Processing the Results

When a SELECT statement is executed, the results of the query will be returned in a ResultSet object defined in the JDBC package. The tuples in the ResultSet can be retrieved using the next() method defined in the ResultSet class. The following code shows how to create and execute a SELECT statement, retrieve the results and display the tuples in the ResultSet object to the screen.

```
String query = "SELECT LastName FROM " +
               "Personnel " +
               "WHERE Department = 'Operations'";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
    String name = rs.getString("LastName");
    System.out.println(name);
}
```

D. JDBC CLASSES AND INTERFACES

1. DriverManager Class

The DriverManager class keeps track of available drivers and handles the creation of connections between databases and appropriate drivers. By invoking the `getConnection()` method of this class, a valid connection with the database can be established.

The DriverManager class contains methods that are used to manage a set of JDBC drivers. Each JDBC driver must provide a class that implements the Driver interface, which is used by the DriverManager. As part of initialization, a program can explicitly tell the DriverManager what driver to load, by using the `Class.forName(<driver name>)` call. If the user does not use this call and attempts to create a connection object, the DriverManager class will check with each registered driver to see if it can connect to the given URL. If more than one driver can connect to the URL, the DriverManager will invoke the first compatible driver encountered. [Ref. 4]

Connection objects are generated from the class DriverManager. When `getConnection()` is called, the DriverManager will attempt to locate a suitable driver from those loaded at initialization and those loaded explicitly using the same class loader as the current applet or application. The URL provided to the `getConnection()` function names the driver to be used to establish the connection. The connection protocol supported by Sun is:

```
jdbc:<subprotocol>://<host>: <portnumber>/<datasource>
```

For example: `String url = "jdbc:dbaw://131.120.1.91:8899/companyDB"` uses jdbc protocol with a dbaw (dbAnywhere) sub protocol to connect to port 8899, on host 131.120.1.91, and then presents the data source name companyDB to the port to locate the specific database. The DriverManager uses this URL to find a registered Driver who can connect to the source.

All DriverManager methods are declared static, which means that they operate on the class as a whole, not on particular instances.

2. Connection Interface

A connection object represents a connection of your application to a database and is used to execute the next phase of database access, creating a statement object, which will allow the user to execute a SQL command. It can also be used to commit a change to the database, as well as rollback.

An application can have one or more connections with a single database or it can have simultaneous connections with multiple databases. The established connection passes SQL statements to the connected database.

3. Statement Interface

There are three statement objects of which the inheritance hierarchy is: Statement, PreparedStatement and CallableStatement. To obtain a statement object the user can call Connection method `createStatement()`. The statement object can be used to execute a SQL statement. This type of statement object is useful for SQL statements that will only

be generated once. There are three types of execute methods that can be used with statement objects:

a. *Execute()*

```
boolean execute(String arg) throws SQLException;
```

This method executes *arg*, which is a SQL statement that may return one or more result sets, one or more update counts, or any combination of these. This method is useful if the designer doesn't know whether the statement will be an update or a query operation. A call to this method executes a SQL statement and returns true if the result is a *ResultSet* and returns false if the result is an update count.

b. *ExecuteQuery()*

```
ResultSet executeQuery(String arg) throws SQLException;
```

This method executes *arg*, which is a SQL statement that returns a single result set representing the results of the provided query.

c. *ExecuteUpdate()*

```
int executeUpdate(String arg) throws SQLException;
```

This method executes a SQL INSERT, UPDATE, or DELETE statement that doesn't have parameter placeholders. It may also be used to execute SQL statements which return no value, such as CREATE TABLE or DROP TABLE.

4. PreparedStatement Interface

If the same SQL statement is executed many times, it is more efficient to use a PreparedStatement. A SQL statement with or without IN parameters can be pre-compiled and stored in a PreparedStatement object. A PreparedStatement object can be more efficient than a Statement object, because it has been pre-compiled and stored by the database. [Ref. 5]

To bind input parameters, setXXX methods are used where XXX can be any primitive type or a String. The setXXX methods for setting IN parameter values must specify types that are compatible with the defined SQL type of the input parameter. For instance, if the IN parameter has SQL type Integer then setInt() should be used. Columns can be referenced by column index, which begin with one, for greater efficiency, or by column name for convenience. The following example demonstrates how a PreparedStatement can be effectively used to populate a table with 100 items, each with a unique partID.

```
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO Parts (partType, partID, quantity) " +
    "VALUES ( ?, ?, ?) ");

pstmt.setString (1, "Tire");
pstmt.setInt(3, 4);
```

```
for(partID = 1; partID <= 100; partID++) {  
    pstmt.setShort(2, partID);  
    pstmt.executeUpdate ();  
}
```

5. CallableStatement Interface

CallableStatement extends PreparedStatement and is used to execute stored procedures. Stored procedures are blocks of SQL code that are stored in the database and executed on the server. This increases efficiency for SQL Statements that are executed often, by reducing the overhead of regenerating an access plan. The DBMS generates and stores the access plan once, and other applications can use the procedure. [Ref. 4]

JDBC provides a stored procedure SQL escape that allows stored procedures to be called in a standard way for all RDBMS's. This escape syntax has one form that includes a result parameter and one that does not. If used, the result parameter must be registered as an OUT parameter. The other parameters may be used for input, output or both.

6. ResultSet Interface

A ResultSet object provides access to a table of data generated by executing a SQL statement. Table rows are retrieved in sequence. Within a row, column values can be accessed in any order. The object maintains a cursor that points to the current row of data, which can be traversed via the next() method. The method next() moves the cursor to the next row, and returns true if the row exists. ResultSet class provides methods that allow access to the results of a query.

SQL Data Type	JAVA data type	Recommended getXXX
CHAR	String	getString()
VARCHAR	String	getString()
LONGVARCHAR	String	getAsciiStream() getUnicodeStream()
NUMERIC	java.math.BigDecimal	getBigDecimal()
DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	boolean	getBoolean()
TINYINT	byte	getByte()
SMALLINT	short	getShort()
INTEGER	int	getInt()
BIGINT	long	getLong()
REAL	float	getFloat()
DOUBLE	double	getDouble()
FLOAT	double	getDouble()
BINARY	byte[]	getBytes()
VARBINARY	byte[]	getBytes()
LONGVARBINARY	byte[]	getBinaryStream()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp()

Table 4.1: SQL and Java Data Types and Recommended Conversion Methods

A `ResultSet` is automatically closed by the statement that generated it when that Statement is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results. Various `getXXX()` methods can be invoked to retrieve different column values. The SQL data types, the corresponding Java data types, and the recommended method calls for conversion are shown in Table 4.1.

7. `ResultSetMetaData` Interface

The interface `ResultSetMetaData` provides information about the types and properties of the columns in a `ResultSet` object. An instance of `ResultSetMetaData` actually contains the information, and `ResultSetMetaData` methods give access to that information.

The following code fragment, where *stmt* is a `Statement` object, illustrates creating a `ResultSetMetaData` object:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM  
Parts"); ResultSetMetaData rsmd = rs.getMetaData();
```

The `ResultSetMetaData` `getMetaData()` method returns a `ResultSetMetaData` object which can provide detailed information about the `ResultSet`, to include column information. This information is useful in presenting the `ResultSet` in an interface.

8. DatabaseMetaData Interface

The interface `DatabaseMetaData` provides information about a database as a whole. One creates an instance of `DatabaseMetaData` and then uses that instance to call methods that retrieve information about a database.

A `DatabaseMetaData` object is created with the `Connection` method `getMetaData`, as in the following code, where *con* is a `Connection` object:

```
DatabaseMetaData dbmd = con.getMetaData();
```

The variable *dbmd* contains a `DatabaseMetaData` object that can be used to get information about the database to which *con* is connected. This is done by calling a `DatabaseMetaData` method on *dbmd* as in the following code fragment:

```
int length = dbmd.getMaxTableNameLength();
```

Many of the `DatabaseMetaData` methods return lists of information in `ResultSet` objects. Data is retrieved from these `ResultSet` objects using the normal `ResultSet` `getXXX` methods, such as `getString()` and `getInt()`.

E. JDBC AND CLIENT/SERVER MODELS

The JDBC API supports database access, utilizing both two-tier and three-tier models. In the two-tier model, a Java applet or application communicates directly with the database. This requires a JDBC driver that can communicate with the particular DBMS being accessed. The user's SQL statements are delivered to the database and the results of the statement are returned to the user. The database may be located on a remote machine to which the user is connected via a network. This is referred to as a *Client/Server Configuration*, with the user's machine as the client, and the machine hosting the database as the server. The network in question can be an Intranet or an Internet.

One issue that makes the client/server model attractive is that an organization can store its business logic, a set of rules that enforce or implement an organization's policies, on a central server. As with all corporate policies, the rules may change. Also, since the business logic can be fairly complex and lengthy, enforcing it in the client will make the client code very large. So, by encapsulating all business logic on the server, organizations can store and change the rules at one location, which reduces administrative costs.

Figure 4.2 describes the two-tier model using JDBC.

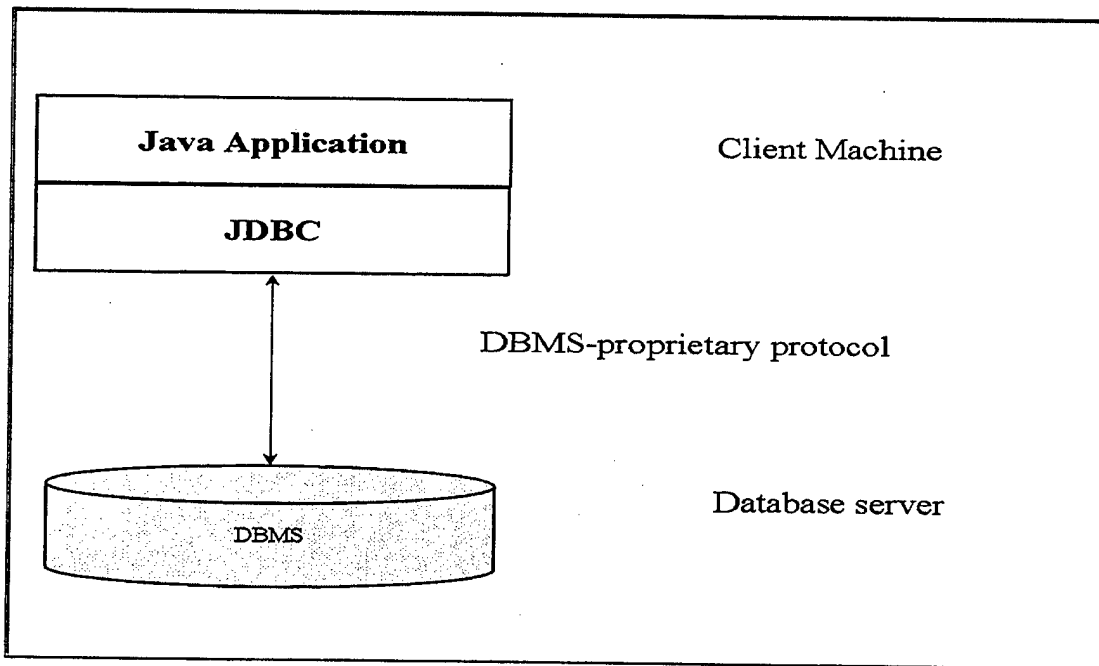


Figure 4.2: JDBC Two-Tier Model [Ref: 10]

In the three-tier model, commands are sent to a middle tier of services, which then sends SQL statements to the database. The database processes the SQL statements and sends the results back to the middle tier, which then passes them to the user. This makes the three-tier model very attractive, because the middle tier makes it possible to maintain control over access and the type of updates that can be made to the data. Another advantage is that when there is a middle tier, the user can employ an easy-to-use higher-level API, which is translated by the middle tier into appropriate low-level calls. The client can operate as a multi-threaded application and let the intermediate server handle the synchronization. In a three-tier architecture the middle layer can hide information about the database server from the applet. Only the middle tier knows how to find and manipulate the data. The secure intermediate server can provide the means to shield the

client from direct access to DBMS by providing a username and password because the identification and authentication is accomplished at the server level. Figure 4.3 describes the three-tier model using JDBC.

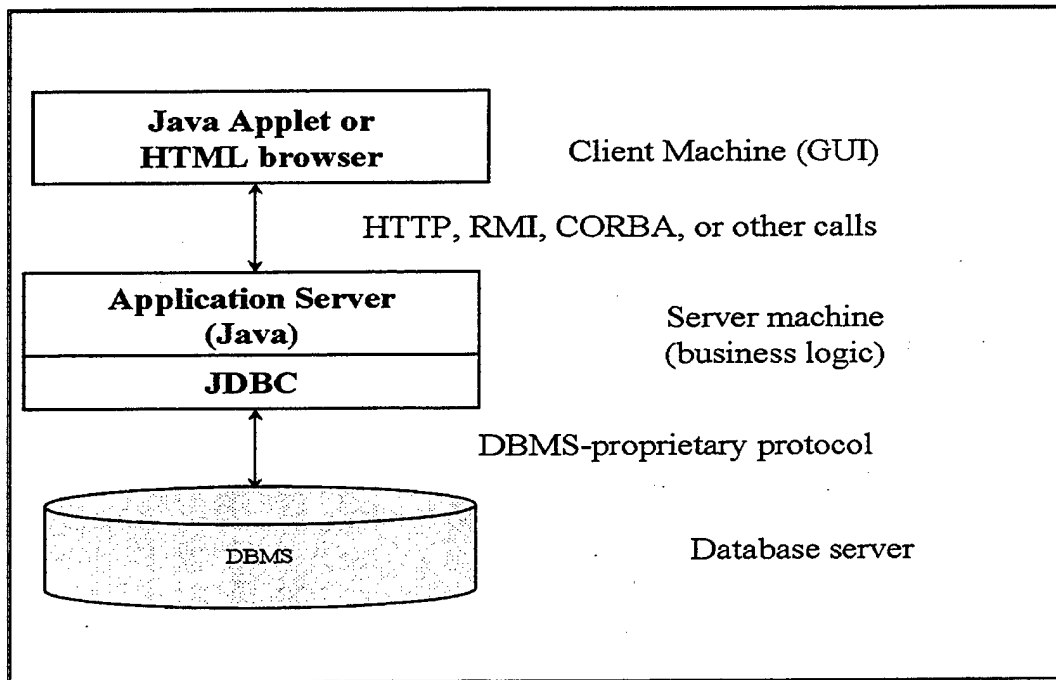


Figure 4.3: JDBC Three-Tier Model [Ref: 10]

F. JDBC DRIVERS

Database drivers provide the implementation of the abstract classes provided by the JDBC API. The driver resides on the Java client machine and is used to establish a connection to a relational database. The JDBC driver can be a JDBC/ODBC bridge, a middleware protocol library, or a native database driver. The driver provides the interface that accepts JDBC input from the Java application, and understands the vendor specific relational database language and network protocols. It accepts the JDBC input from the

client application, translates it to a vendor specific protocol, and uses a vendor supporting networking protocol to transmit the request across the network. [Ref: 4]

Figure 4.4 depicts the JDBC driver implementation.

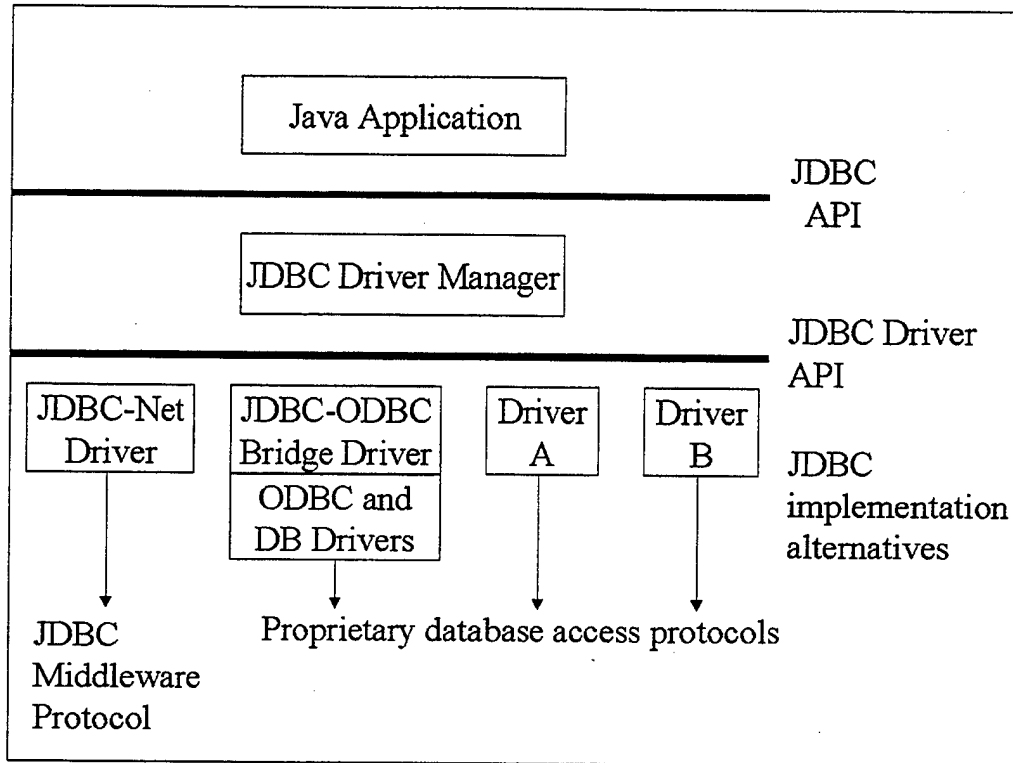


Figure 4.4: JDBC Driver Implementation [Ref: 5]

There are four categories of JDBC drivers as designated by JavaSoft. [Ref. 12]

The following are the various classes of JDBC drivers available:

1. JDBC-ODBC Bridge plus ODBC Driver

The JDBC/ODBC Bridge was designed to take advantage of the large number of ODBC enabled drivers. The bridge was intended to provide an initial solution until database vendors could produce their own vendor specific JDBC drivers. Basically the

bridge converts the JDBC calls into ODBC calls. The ODBC driver manager, will invoke the database vendor specific ODBC driver, and pass the calls to database driver for further processing.

The client side application or applet uses the JDBC API to load the "sun.jdbc.odbc.JdbcOdbcDriver". This driver translates the Java SQL statements into ODBC format, then invokes the ODBC Driver Manager (odbc32.dll) which refers to the odbc.ini file that contains a data source name and vendor specific driver it is associated with. The vendor specific driver, or DLL, then translates the ODBC call into a vendor specific call, and sends the request across the network to the database manager. The process is reversed when a response is sent from the database manager back to the client application. [Ref. 4]

2. Native-API partly-Java Driver

This type of driver converts JDBC calls on the client API to a vendor specific query language and communication protocol for use on a DBMS [Ref. 5]. The drivers are usually written in C, accept the Java calls, then map them to vendor specific calls. The call then gets processed by the vendor specific driver, translating it into the DBMS's specific query language and communication protocol. This is a partly Java driver, that requires a vendor supplied library to translate JDBC functions into the DBMS's specific query language, such as Oracle's OCI [Ref. 4].

3. JDBC-Net pure Java Driver

The JDBC-Net pure Java Driver translates JDBC calls into a database independent network protocol, and passes this request to a middle tier server, which then translates the request into a DBMS specific protocol. These drivers are attractive for Internet/Intranet based multi-user data intensive applications (requiring access to multiple databases). The JDBC-Net pure Java Driver contains a number of vendor specific drivers, or can use the ODBC/JDBC driver to provide database access. The bridge can connect the client to local databases, which must reside on the same machine as the middleware server, such as MS Access, or to remote databases such as Oracle, MS SQL Server, SyBase, InterBase, or IBM DB/2, stored on another machine. As can be expected, this driver is slower than other JDBC drivers. However, this is the most flexible of all driver implementations. [Ref. 4]

4. Native-protocol pure Java Driver

Native-protocol pure Java drivers convert the JDBC calls directly into the network protocol used by the specific DBMS. This allows a direct call from the client machine to the DBMS server and is a practical solution for Internet access. These drivers can be written entirely in Java, and can provide just in time delivery of applets. The Native-protocol pure Java drivers provide for the best database access because of the direct translation, unfortunately they can only be supplied by the vendor and can only interface with the vendor specific database. For example, Sybase jConnect is a Native-protocol pure Java JDBC driver written entirely in Java and communicates directly to Sybase data sources such as Sybase SQL Anywhere. Since many of these protocols are

proprietary in nature, the driver can only interface with vendor specific databases. The most significant advantage of this driver is its speed, where the biggest disadvantage is the loss of flexibility [Ref. 5].

5. Driver Selection

The selection of which type of driver to employ depends upon a number of factors: number of databases requiring access, performance requirements, financial, and system administration requirements. In the POET Database Application Program, I am using the JDBC-ODBC Bridge plus ODBC Driver, which translates the Java SQL statements into ODBC format and invokes the ODBC Driver Manager.

A JDBC/ODBC bridge is effective for an application server. A middleware server provides the database access, so all ODBC drivers reside on that machine. The client makes a call to the application server, which establishes the database connection and returns a string or data stream to the client.

THIS PAGE INTENTIONALLY LEFT BLANK

V. REQUIREMENTS ANALYSIS FOR POET DATABASE

This chapter provides information about both the database development process and the requirements analysis for the POET database system.

A. DATABASE DEVELOPMENT PROCESS

The database development process described here consists of four phases: *requirements collection and analysis*, *conceptual database design*, *logical database design*, and *physical database design*.

1. Requirements Collection and Analysis

The first step in the database development process is *requirements collection and analysis*. Requirements collection and analysis phase constitute the most important step of the entire database design process, because most subsequent design decisions are based on this step. The major task is collecting information content and processing requirements from all the identified and potential users of the database. Analysis of the requirements ensures the consistency of users' objectives as well as the consistency of their views of the organization's information flow.

During this step, the database designers interview prospective database users to understand and document their data requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known functional requirements of the application. These consist of the user-defined operations that will be applied to the

database, and they include both retrievals and updates. [Ref. 1]

The purpose of this phase is to determine, as specifically as possible, what the system must do. There are two tasks in this phase:

- Specify the data requirements.
- Determine the functional requirements.

a. Data Requirements

During the data requirements phase, the major goals are to build a data model that documents the "things" that are to be represented in the database, to determine the characteristics of those "things" that need to be stored and to determine the relationships among them. The user's data model describes the objects that must be stored in the database, along with their structure and the relationships that they have with one another. The output of the data requirements phase is a statement of requirements. This statement can take a variety of forms: a verbal description, an entity-relationship diagram, semantic object diagram, one or more prototypes, or any combination of the above.

The "things" that are represented in the database are referred to as either entities or semantic objects, depending on the modeling technique that the designer follows. In this thesis, the semantic data model will be used as the high-level modeling technique. Semantic Data Model was described in Chapter III.

b. Data Dictionary

A *data dictionary* is a catalog of requirements and specifications for a new information system. It provides definitions of all the data items in the database. During the definition phase, the analysts try to capture and store the data in the system, and find the inputs and outputs that the system will generate. These are represented with pictorial models such as data flow diagrams, relation diagrams, entities, data stores, etc. The data dictionary expands this pictorial model and as a system analysis tool, captures the detailed requirements for every input, output and data store. The suggested approach for building the data dictionary should be in terms of "*what*" data are handled and not in terms of "*how*" data are presented or formatted.

c. Process Requirements

All systems process data to produce information and maintain stored data. These requirements should be logically modeled. In order to implement processes as programs, a process model is needed. A process model is a picture of the flow of data through the system and the processing that must be performed on that data. These processes interact or interface with one another. These interactions take the form of data flows between processes and is the reason that they are sometimes called *data flow models*.

2. Conceptual Database Design

Once all the requirements have been collected and analyzed, the next step is to create a conceptual schema for the database, using a high-level conceptual data model, such as Entity-Relationship Model or Semantic Data Model. This step is called *conceptual database design*. The conceptual schema is a concise description of the data requirements of the user and includes detailed descriptions of the data types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include any implementation details, they are usually easier to understand and can be used to communicate with non-technical users. The high-level conceptual schema can also be used as a reference to ensure that all of the requirements are met and that the requirements do not include any conflicts. This approach enables the database designers to concentrate on specifying the properties of the data, without being concerned with storage details. [Ref. 1]

The main purpose of conceptual design is to represent information in a form that is comprehensible to the user, independent of system specifics, but implementable on several systems. The result of conceptual design is called the conceptual schema, because it is a representation of the user's "world" view and independent of any DBMS software or hardware considerations.

In order to build an effective database and related applications, a data model that captures the users' perceptions closely is of great importance. The data model should identify the entities and their attributes to be stored in the database and should define their structure and the relationships among them.

After the conceptual schema has been designed, the basic data model operations can be used to specify high-level transactions corresponding to the user-defined operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements can not be specified in the initial schema.

3. Logical Database Design

The next step in the database design is the actual implementation of the database, using a commercial DBMS, such as Oracle, Sybase, Informix, DB2, or Access. The major goal of the logical database design phase is to use the results of the conceptual design phase and the processing requirements as input to create a DBMS-processible schema as output.

Most currently available commercial DBMSs use an implementation data model, such as Relational, Network, Hierarchical, or Object-Oriented, so the conceptual schema is transformed from the high level data model into the implementation data model. This step is called *logical database design*, and its result is a database schema in the implementation data model of the DBMS. [Ref. 1]

After the semantic objects are developed in the conceptual database design phase, these objects are transformed into an implementation data model during this phase. In this thesis, the relational model, which is the most common data model used in commercial DBMSs, will be used as the implementation data model. Therefore, the semantic objects will be transformed into relations as described in Chapter III. After the relations are

created, they are then normalized. This is a very important part of the design, because we need to be sure that the relations will not suffer from any update anomalies. The process of normalization was discussed in Chapter II.

4. Physical Database Design

Finally, the last step is the *physical database design* phase, during which the internal storage structures and file organizations for the database are specified. Physical database design is the process of developing an efficient and implementable physical database structure from a given logical database structure that has been shown to satisfy user information requirements.

In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. An application is the collection of menus, forms, reports, and programs that provide a means of update, display, and control the objects of the data model. During the application design, the specific structure of forms, reports, menus, and query facilities are defined.

The application program for the POET database system is developed by using Java, an object-oriented programming language, and JDBC, an Application Program Interface (API) that allows a Java program to communicate with a database server using Structured Query Language (SQL) commands. Chapter IV describes Java programming language and JDBC Application Program Interface.

B. REQUIREMENTS ANALYSIS FOR POET DATABASE

Data requirements for the POET database system are captured in the form of semantic objects and associated data dictionary. This application consists of seven semantic objects, which are *ship*, *department*, *division*, *personnel*, *operation*, *equipment*, and *training*, and the semantic object diagrams of these objects are shown in Appendix A.

1. Ship Object

The ship object represents the frigate on which the POET database system will be installed. The ship is uniquely identified by its international call sign. Ship object has International Call Sign, Ship Name, Hull Number, Ship Class, Keel Laying Date, Launch Date, and Commission Date simple attributes that provide information about the ship's identity and history. It also has Length, Width, Mast Height, Keel Depth, and Displacement simple attributes that describe the ship's physical dimensions.

In order to keep track of the overhaul information, multivalued group attribute Overhauls containing simple attributes Overhaul Number, Start Date, End Date, Shipyard Name, and Overhaul Duration is placed within the ship object. The ship belongs to a higher command, and the attribute Immediate-Superior-In-Command keeps this data. Another attribute of the ship is the name of its Homeport.

The group attribute Planned Manning stores the number of the personnel that should be stationed on the ship and it has the attributes Planned Officers, Planned Petty Officers, and Planned Enlisted. Present Manning group attribute, on the other hand, stores the actual number of the personnel currently stationed onboard and it has the attributes Present Officers, Present Petty Officers, and Present Enlisted.

2. Department Object

The Turkish Navy frigates are organized into six departments; Operations, Engineering, Weapons, Electronics, Navigation, and Supply. The Department object represents the departments within the ship and stores the department specific information needed by the command. A department is uniquely identified by its name and it is controlled by the Department Head, who is an officer. Department object has the simple attribute Department Name, and group attributes Planned Manning and Present Manning. Every department contains zero or more DIVISIONs and one or more PERSONNEL.

3. Division Object

The departments on the Turkish Navy frigates are organized into divisions. The Division object represents the divisions within the ship and stores the division specific information needed by the command. Similar to a department, a division is uniquely identified by its name and it is controlled by the Division Officer. Division object has the simple attribute Division Name, and group attributes Planned Manning and Present Manning. Every division contains one or more PERSONNEL and it belongs to one and only one DEPARTMENT.

4. Personnel Object

The Personnel object represents the crewmembers who consist of officers, petty officers, and enlisted people stationed on the ship. Every person has a unique military identification number.

The personnel onboard the ship work for a department and a division under that department; therefore, the Personnel object contains semantic object attributes DEPARTMENT and DIVISION.

Personnel object has the following simple attributes that provide personal information: Military Identification Number, First Name, Last Name, Rank, Rating, Date of Birth, Place of Birth, Father's Name, Mother's Name, Active Duty Service Date, Date of Rank, Sex, Marital Status, Spouse Name, Number of Children, Street, City, State, Zip Code, and Phone Number.

The command is also interested in other personal data, like the person's training, previous assignments, and foreign languages. The multivalued group attribute Courses-To-Take specifies the military courses that the person should take according to his/her career. Courses-Taken, which is another multivalued group attribute, describes the courses taken by the person and it includes TRAINING object, Start Date, End Date, and Grade attributes. The person may have been assigned to zero or more previous duties before the ship, so Previous Assignments group attribute, with the simple attributes Assignment Number, Station, Position, and Duration, stores this information. Language Name and Degree represent the foreign languages that the person knows.

Another useful data about the personnel is the Specialty and background Education. As soon as the person embarks, he/she is assigned a Cabin Number and Cabin Phone Number. Also, the Current Assignment and Start Date of the current duty are stored within the Personnel object.

5. Training Object

The training of crewmembers is of great importance for their career and it is a continual activity. There are required military courses that crewmembers have to attend in order to get promoted and to be assigned to some duties. The ship's command needs to know what courses the personnel have attended, the start date, the end date, and the degree or grade that the person obtained. It is also necessary to keep track of the courses that the person must take in order to perform his/her task successfully.

Each course is uniquely identified by the course name. The Training object has the simple attributes Course Name, Training Center, Course Duration, and Course Description, which provide the needed information by the command about the courses.

6. Operation Object

The data about the operations is one of the most frequently searched information, because most of the reports and messages are related with the ship's operations. The exercises in which the ship has participated constitute the main part of the operations information. The command needs to know the Exercise Name, Exercise Type, Start Date, End Date, Duration of the Exercise, and Place of the Exercise, which are the simple attributes of the Operations object.

It is also required to store data about the specific events executed during the exercises. Therefore, the multivalued group attribute Events serves this purpose, while containing Event Name, Event Type, Event Duration, and Number of Events attributes. Port Visits group attribute keeps the data about the ports that have been visited by the

ship during an exercise. It includes Port Name, Visit Start Date, Visit End Date, and Visit Duration simple attributes.

The command also keeps track of the Underway Hours, separating them as Daytime and Nighttime Underway Hours. Another property of an operation is its cost. So, Cost of Exercise group attribute provide the Fuel Cost, Ammunition Cost, and Amortization Costs.

The frigates generally host helicopters during the operations. Helicopter group attribute, which contains Helo Tail Number, Flying Duration, Number of Dippings, and Dipping Duration, keeps the helicopter information needed by the command.

7. Equipment Object

The ship contains a lot of equipment that have critical role in the functionality of the ship. The ship's immediate-superior-in-command wants to know the status of the ship's equipment, such as which of them are out-of-order, which are operating efficiently, which one has frequent failures. In order to serve this purpose, Equipment object is defined as part of the POET database. Every equipment has a Serial Number, a Stock Number, an Equipment Name, an Equipment Type, a Manufacturer, a Model, a Production Date, and Runtime that shows the total number of operating hours since the equipment's installation. Serial Number is an attribute that can uniquely identify each equipment.

Since the failures constitute a substantial fraction of the necessary information about equipment, a multivalued group attribute, Failures, is included within the Equipment object. Failure Number, Failure Description, Diagnosis, Failure Date, and Failure Duration are the simple attributes contained within Failures group attribute.

C. DATA DICTIONARY FOR POET DATABASE

The data dictionary presents a tabular specification of the POET data model and it consists of two parts: Semantic Object Specifications and Domain Specifications. The semantic objects, their attributes, minimum, and maximum cardinalities are defined in the Semantic Object Specifications table. This table is an alternative presentation of the information provided by the semantic object diagrams, and it is shown in Appendix B.

The Domain Specifications table describes the domains of the objects and attributes. This table, however, supplies information about domains that is not available from the semantic object diagrams. The semantic and physical description of each domain is provided in this table, which is shown in Appendix C.

VI. LOGICAL DATABASE DESIGN FOR POET DATABASE

This chapter discusses the logical database design for the POET database system. In logical database design, the semantic object model developed in the previous chapter is transformed into a relational schema, in preparation for the database implementation using a specific DBMS. The POET database will be implemented in Microsoft Access 97, which is an easy-to-use, affordable, and true relational database management system.

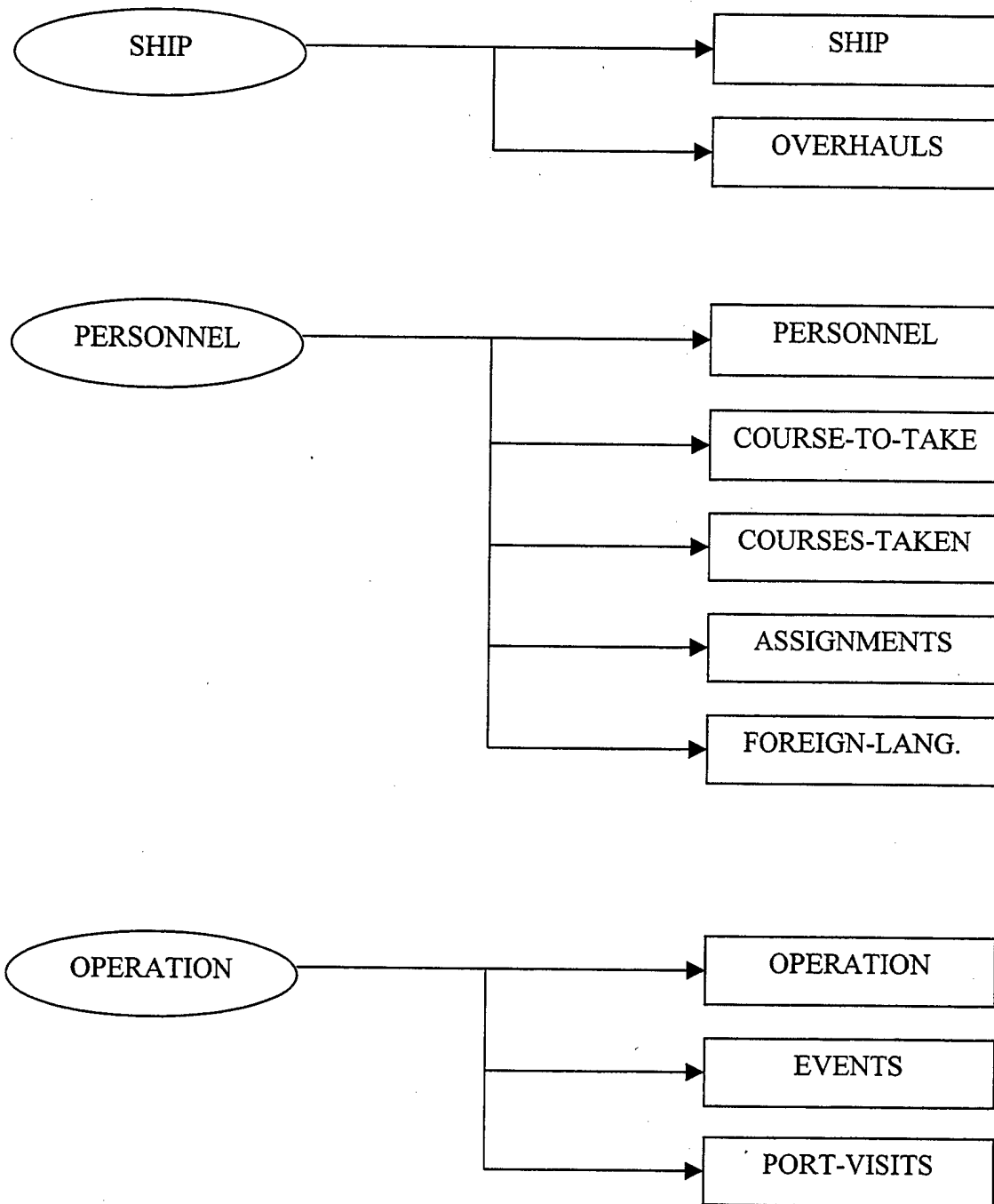
The seven semantic objects describing the personnel, operations, equipment, and training subjects onboard Turkish Navy frigates are transformed into relational tables. The semantic objects are transformed into relations by using the rules described in Chapter III. As a result of the conversion process, fifteen relational tables are obtained. The semantic objects and the corresponding relations that are defined are shown in Figure 6.1. The relationships among the tables are represented using foreign keys and are also shown explicitly on the relational schema. In this diagram, which is shown in Appendix D, primary keys are underlined and made boldface while foreign keys are italicized in order to distinguish them from other attributes. The relations shown in Appendix D, their attributes, and relationships among them are discussed in the following sections.

A. RELATIONAL TABLES OF POET DATABASE

The transformation of seven semantic objects has yielded the following fifteen relations: SHIP, OVERHAULS, PERSONNEL, COURSES-TO-TAKE, COURSES-TAKEN, ASSIGNMENTS, FOREIGN-LANGUAGES, DEPARTMENT, DIVISION, TRAINING, OPERATION, EVENTS, PORTVISITS, EQUIPMENT, and FAILURES.

SEMANTIC OBJECT

RELATIONAL TABLE



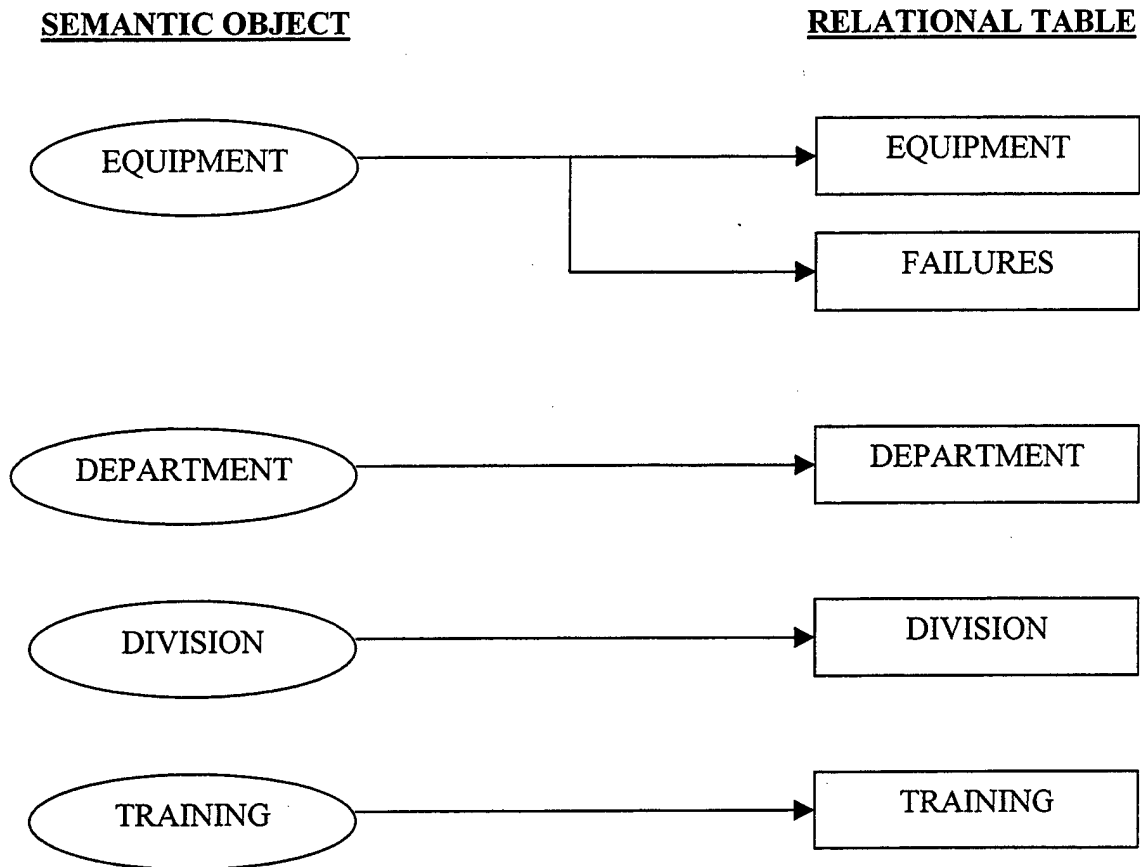


Figure 6.1: Semantic Object – Relational Table Transformation

1. Ship Relation

The SHIP relation contains information about the frigate on which the POET database system will be installed and it is derived from the SHIP object. Each attribute of the semantic object is defined as an attribute of the relation, and the identifying attribute, International Call Sign, becomes the primary key of the relation.

The SHIP table consists of the following attributes: International Call Sign, Ship Name, Hull Number, Ship Class, Keel Laying Date, Launch Date, Commission Date, Length, Width, Mast Height, Keel Depth, Displacement, Homeport, Immediate Superior In Command, Planned Officers, Planned Petty Officers, Planned Enlisted, Present Officers, Present Petty Officers, and Present Enlisted.

2. Overhauls Relation

The OVERHAULS relation provides the necessary data needed by the command about the overhauls of the ship. This relation is derived from the SHIP object. Since Overhauls is a multivalued group attribute of the SHIP object, a new relation is defined in order to translate the semantic data model correctly into relational model. The primary key of the OVERHAULS table is International Call Sign and Overhaul Number, which is the composite of the identifier of the SHIP object plus the identifier of the Overhauls group attribute.

The other attributes of this relation are Start Date, End Date, Shipyard Name, and Overhaul Duration. There is a one-to-many relationship between the SHIP table and the OVERHAULS table, because the ship may have zero or more overhauls.

3. Department Relation

This relation contains information about the departments of the ship. It is derived from the DEPARTMENT object. Each attribute of the semantic object is defined as an attribute of the relation. The primary key of the DEPARTMENT table is Department Name, and the other attributes are Planned Officers, Planned Petty Officers, Planned

Enlisted, Present Officers, Present Petty Officers, and Present Enlisted.

Every department contains zero or more divisions, so the DEPARTMENT table has a one-to-many relationship with the DIVISION table. It has also one-to-many relationship with the PERSONNEL table, because each department has one or more crewmembers working for it.

4. Division Relation

The DIVISION relation contains information about the divisions, the organizational unit under the departments. It is derived from the semantic object, DIVISION. Each attribute of the semantic object is defined as an attribute of the relation, and the identifying attribute, Division Name, becomes the primary key of the relation. It also contains the foreign key, Department Name, which establishes the 1:M relationship between DEPARTMENT and DIVISION tables.

Planned Officers, Planned Petty Officers, Planned Enlisted, Present Officers, Present Petty Officers, and Present Enlisted are the other attributes contained in this relation. Similar to the DEPARTMENT table, it has one-to-many relationship with the PERSONNEL table.

5. Personnel Relation

The PERSONNEL relation contains information about the ship's crewmembers, including the officers, petty officers, and the enlisted. It is derived from the PERSONNEL semantic object. The primary key of this relation is the Military Identification Number.

The PERSONNEL table consists of the following attributes that provide personal information: Military Identification Number, First Name, Last Name, Department Name, Division Name, Rank, Rating, Date of Birth, Place of Birth, Father's Name, Mother's Name, Active Duty Service Date, Date of Rank, Gender, Marital Status, Spouse Name, Number of Children, Street, City, State, Zip Code, Phone Number, Specialty, Education, Current Assignment, Start Date, Cabin Number, and Cabin Phone Number.

This relation has many-to-one relationships with DEPARTMENT and DIVISION relations, because a crewmember works for a department and a division under that department. For each multivalued group attribute of the PERSONNEL object, a new table is defined. Therefore, COURSES-TO-TAKE, CORSES-TAKEN, ASSIGNMENTS, and FOREIGN-LANGUAGES tables are created. Since all of these tables represent specific personal information, the PERSONNEL table has one-to-many relationships with COURSES-TO-TAKE, CORSES-TAKEN, ASSIGNMENTS, and FOREIGN-LANGUAGES relations.

6. Courses-To-Take Relation

This relation provides information about the military courses that the personnel should take. It is derived from the PERSONNEL object, because Courses-To-Take is a multivalued group attribute of the PERSONNEL semantic object. The primary key of the COURSES-TO-TAKE table is Military Identification Number and Course Name, which is the composite of the identifier of the PERSONNEL object plus the identifier of the TRAINING object. This relation does not include any attributes, but the primary key.

COURSES-TO-TAKE table has many-to-one relationships with PERSONNEL and TRAINING tables. It can also be viewed as an association table between PERSONNEL and TRAINING tables, that converts a many-to-many relationship to two one-to-many relationships.

7. Courses-Taken Relation

COURSES-TAKEN relation contains information about the military courses that the personnel have previously taken. Like COURSES-TO-TAKE relation, it is derived from the PERSONNEL object, because Courses-Taken is a multivalued group attribute of the PERSONNEL semantic object. The primary key of the CORSES-TAKEN table is Military Identification Number and Course Name, which is the composite of the identifier of the PERSONNEL object plus the identifier of the TRAINING object. Unlike the COURSES-TO-TAKE table, this relation also includes three more attributes, which are Start Date, End Date, and Grade.

COURSES-TAKEN table has many-to-one relationships with PERSONNEL and TRAINING tables. It can also be viewed as an association table between PERSONNEL and TRAINING tables, that converts a many-to-many relationship to two one-to-many relationships.

8. Assignments Relation

This relation provides information about the previous assignments of the personnel. ASSIGMENTS relation is derived from the PERSONNEL object, because Assignments is a multivalued group attribute of the PERSONNEL semantic object and

for each multivalued attribute, a new relation must be defined. The primary key of the ASSIGNMENTS table is Military Identification Number and Assignment Number that is composed of the identifier of the PERSONNEL object plus the identifier of the Assignments group.

The other attributes of the relation are Station, Position, and Duration. ASSIGNMENTS table has a many-to-one relationship with PERSONNEL table.

9. Foreign-Languages Relation

FOREIGN-LANGUAGES relation contains information about which foreign languages are known by which crewmembers. It is derived from the PERSONNEL object, because Foreign-Languages is a multivalued group attribute of the PERSONNEL semantic object. The identifier of the PERSONNEL object and the identifier of the Foreign-Languages group constitute the primary key of the FOREIGN-LANGUAGES table. Hence, the primary key is Military Identification Number plus Language name. It also includes the attribute Degree, which shows the status of the person's language level.

FOREIGN-LANGUAGES table has a many-to-one relationship with PERSONNEL table.

10. Training Relation

This relation stores information about the military courses that are related with the personnel's training. It is derived from the TRAINING semantic object. The primary key of TRAINING relation is Course Name.

Other attributes of this relation are Training Center, Course Duration, and Course Description. TRAINING table has one-to-many relationships with COURSES-TO-TAKE and COURSES-TAKEN tables, because both of these two tables include the Course Name, the primary key of TRAINING table, as foreign key.

11. Operation Relation

The OPERATION relation contains information about the operations and exercises that the ship has participated in. This relation is derived from the OPERATION semantic object. Exercise Name is the primary key of OPERATION relation.

Exercise Type, Start Date, End Date, Duration, Place, Daytime Underway Hours, Nighttime Underway Hours, Helo Tail Number, Flying Duration, Number of Dippings, Dipping Duration, Fuel Cost, Ammunition Cost, Amortization, and Cost of Exercise are the other attributes that are included in the OPERATION table.

OPERATION relation has one-to-many relationships with PORT-VISITS and EVENTS relations, because during an operation, zero or more ports may be visited and zero or more events may be executed.

12. Events Relation

This relation provides information about the events that are performed during exercises and operations. EVENTS relation is derived from the OPERATION object, because Events is a multivalued group attribute of the OPERATION semantic object and for each multivalued attribute, a new relation must be defined. The primary key of the EVENTS table is Exercise Name and Event Name that is composed of the identifier of

the OPERATION object plus the identifier of the Events group.

The other attributes of the relation are Event Type, Number of Events, and Event Duration. EVENTS table has a many-to-one relationship with the OPERATION table.

13. Port-Visits Relation

The PORT-VISITS relation contains information about the port visits that are carried out during exercises and operations. Similar to EVENTS relation, this relation is also derived from the OPERATION object, because Port-Visits is a multivalued group attribute of the OPERATION object. The identifier of the OPERATION object and the identifier of the Port-Visits group constitute the primary key of the PORT-VISITS table. Therefore, the primary key is Exercise Name and Port Name.

The PORT-VISITS relation also includes Visit Start Date, Visit End Date, and Visit Duration attributes. PORT-VISITS table has a many-to-one relationship with the OPERATION table.

14. Equipment Relation

The EQUIPMENT relation provides information about the equipment onboard the ship and their status. It is derived from the EQUIPMENT semantic object. The primary key of the EQUIPMENT relation is Serial Number, a unique identifier for every item of equipment.

Serial Number, Stock Number, Equipment Name, Equipment Type, Manufacturer, Model, Production Date, Location and Runtime are the other attributes that are included in the EQUIPMENT table.

EQUIPMENT table has a one-to-many relationship with FAILURES table, because equipment may have zero or more failures.

15. Failures Relation

This relation stores information about the equipment failures. It is derived from the EQUIPMENT object, because Failures is a multivalued group attribute of the OPERATION semantic object. The primary key of the FAILURES table is the combination of Serial Number, the identifier of the EQUIPMENT object and Failure Number, the identifier of the Failures group.

The other attributes of the FAILURES table are Failure Description, Failure Diagnosis, Failure Date, and Failure Duration. FAILURES table has a many-to-one relationship with the EQUIPMENT table.

B. POET DATABASE RELATIONSHIPS

The relationships among the fifteen relational tables described in the previous section are shown in the following figure.

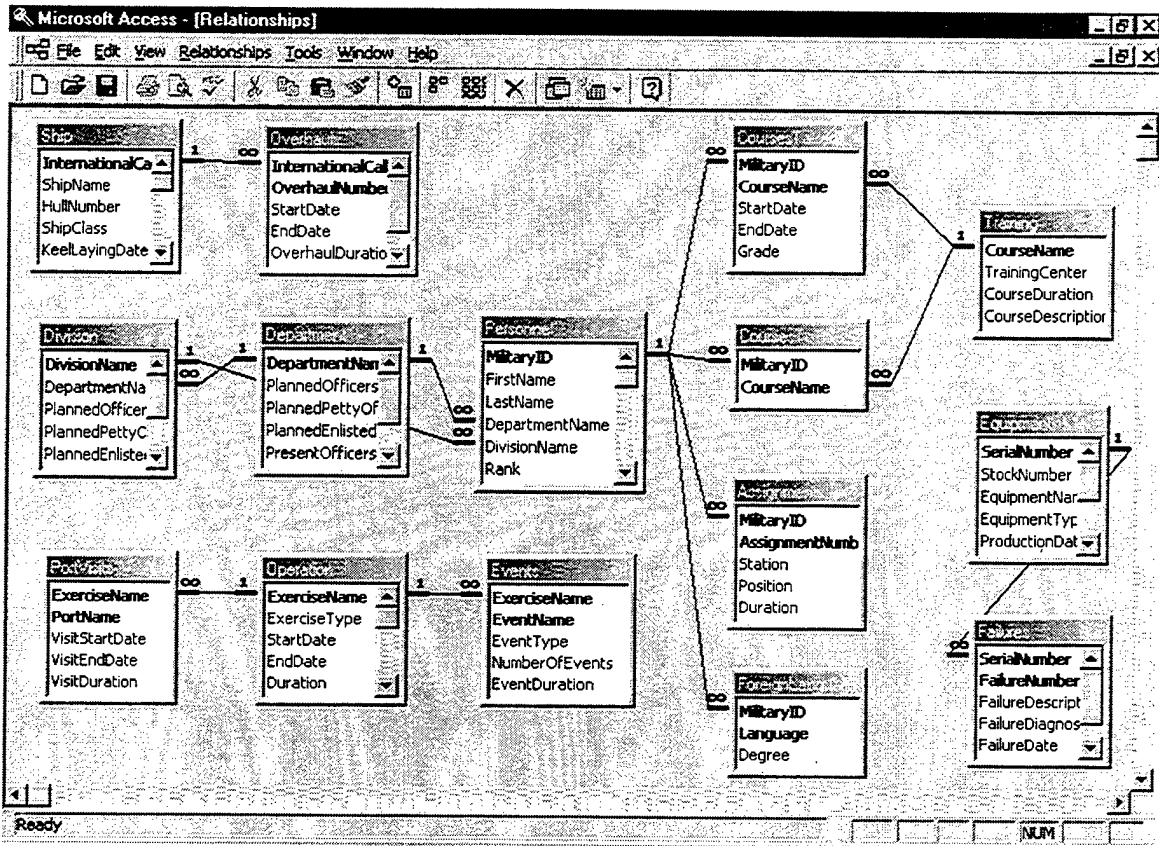


Figure 6.2: POET Database Relationship Diagram Expressed in Relational Database Referential Integrity Constraints

VII. IMPLEMENTATION OF POET DATABASE AND DEVELOPMENT OF APPLICATION PROGRAM

This chapter will discuss the implementation issues for both POET database and the application program.

A. POET DATABASE IMPLEMENTATION

In POET database implementation, the relations and their attributes developed during the logical database design are transformed into tables and data fields, respectively. As stated earlier, Microsoft Access database management system is used as the DBMS of choice for POET database implementation. Therefore, the relational tables are created in Microsoft Access by choosing *Tables/New/Design View* from the database dialog box.

The structure of the new empty table, which matches the corresponding relation developed during the design phase, is then specified. For each attribute of the relation, field name, data type, and optional description are entered. After the field name and data type are described, by using the field properties section of the table design grid, more specific properties can be defined; such as Field Size, Format, Input Mask, Caption, Default Value, Validation Rule, Validation Text, Required, Allow Zero Length, Indexed, and Lookup. Upon definition of all attributes of the relation, the primary key is specified and the new table is saved by giving it a name; i.e., *Personnel*.

Brief descriptions of the field type choices are displayed on the table design grid to assist the user in creating the new table. The data types supported by Microsoft Access are shown in Figure 7.1.

Data Type	Type of Data Stored	Storage Size
Text	Alphanumeric characters	0-255 characters
Memo	Alphanumeric characters	0-64,000 characters
Number	Numeric Values	1, 2, 4, or 8 bytes
Date/Time	Date and Time	8 bytes
Currency	Monetary data	8 bytes
Auto Number	Automatic number increments	4 bytes
Yes/No	Logical values: True/False	1 bit (0 or 1)
OLE Object	Pictures, graphs, sound, video	Up to 1GB
Hyperlink	Link to an Internet source	0-6,144 characters
Lookup Wizard	Displays data from another table	Generally 4 bytes

Figure 7.1: Data Types Available in Microsoft Access

Once the definition of a table is completed, the user can enter values in the table directly in datasheet format or through a form. In this implementation, the user will be using data entry forms provided by the application program, instead of MS Access forms. The following figure shows the table definition for Operation relation.

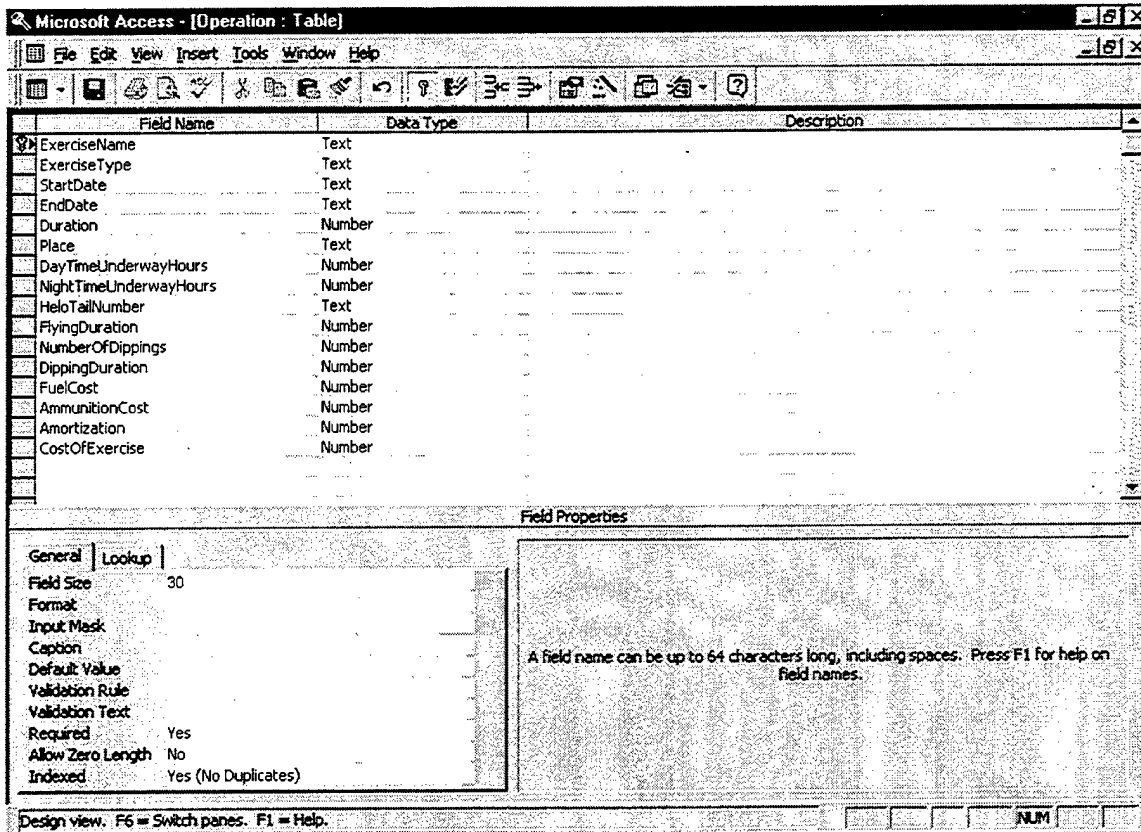


Figure 7.2: Table Design View for Operation Relation

POET database also includes thirteen predefined queries that will support the reports and the static queries of the application program. These queries are created with the Access method Query by Example (QBE). In this method, the user first selects the tables that will be used in the query. When the user enters instructions into the QBE window, Access translates them into SQL statements and retrieves the desired data by filtering the records, selecting only those meeting the query criteria. Figure 7.3 shows the QBE window for the Previous Assignments Query.

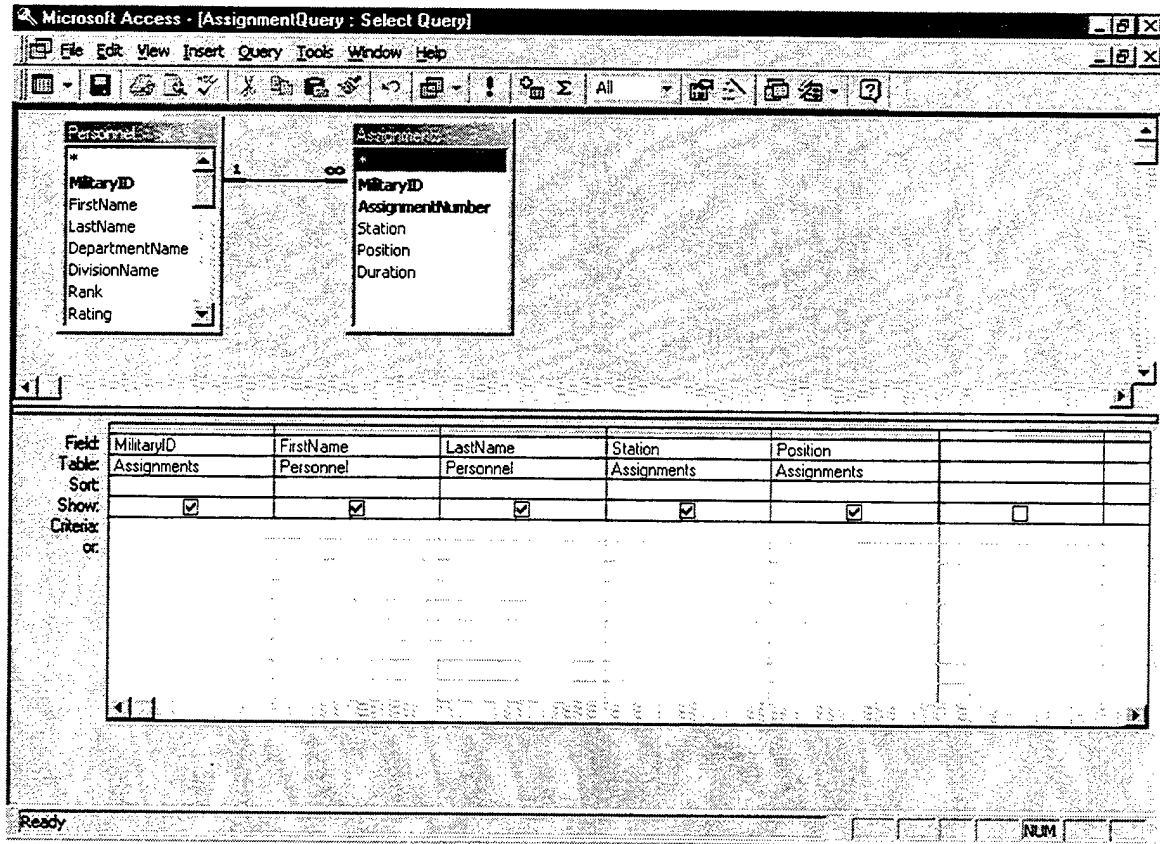


Figure 7.3: QBE Window for Previous Assignments Query

B. APPLICATION PROGRAM IMPLEMENTATION

The purpose of developing an application program is to allow the ship personnel to access the information in a windows-based environment without the need of a database management system environment, and thus to eliminate the need for learning a database management system. The architecture of the POET application program is shown in Figure 7.4.

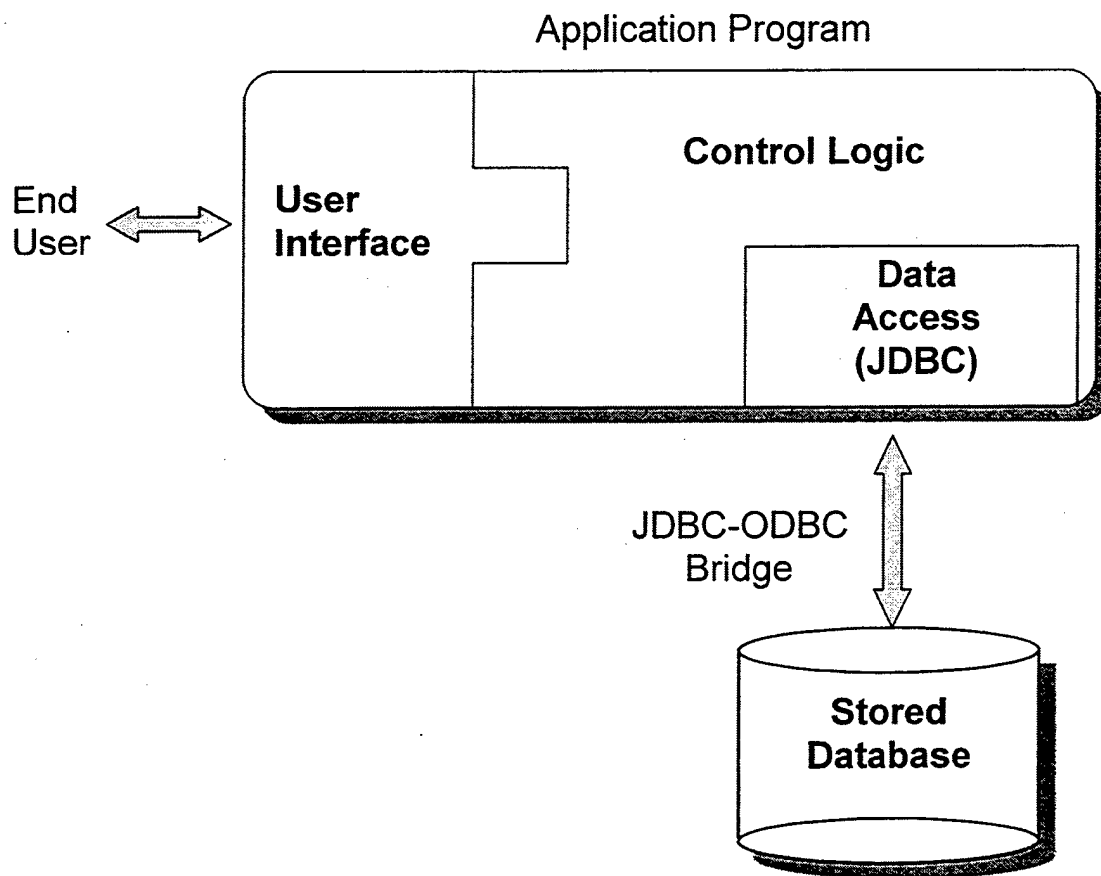


Figure 7.4: POET Application Program Architecture

The application program is developed by using Java programming language and Java Database Connectivity (JDBC), an Application Program Interface (API) that allows a Java program to communicate with a database server using Structured Query Language (SQL) commands. JDBC provides the object-oriented application program the ability to communicate with Microsoft Access relational database management system via JDBC-ODBC Bridge. Java and JDBC were described in Chapter IV. The complete Java code for the POET application program is included in Appendix F.

The application program consists of a graphical user interface (GUI) and a control logic that allows the users to access the data stored in MS Access relational DBMS. It provides data input forms, data update forms, tables, reports, and queries that are similar to the ones supported by MS Access. The following section will discuss each of these components in detail.

1. Input Forms

Data Input Forms are used to add new records into a database table in a quick, easy, and accurate manner. POET application program includes the following input forms. Figure 7.5 shows one of these forms, Operation Input Form.

- Personnel Input Form
- Operation Input Form
- Equipment Input Form
- Training Input Form
- Overhaul Input Form
- Courses-To-Take Input Form
- Courses-Taken Input Form
- Previous Assignments Input Form
- Foreign Languages Input Form
- Event Input Form
- Port Visit Input Form
- Failure Input Form

OPERATION FORM

Exercise Name :

Exercise Type : SQUADRON EXERCISE ▼

Start Date :

End Date :

Duration (Days) :

Place (Sea/Ocean) :

Daytime Underway Hours :

Nighttime Underway Hours :

Helo Tail Number :

Helo Flying Time (Hours) :

Number Of Dippings :

Total Dipping Time (Hours) :

Fuel Cost :

Ammunition Cost :

Amortization :

Cost Of Exercise :

ADD RECORD DELETE RECORD UPDATE RECORD CANCEL

Figure 7.5: Operation Input Form

2. Update Forms

Data Update Forms are the user's primary interface for modifying and deleting records in a table. POET application provides the same number of update forms with the same names as input forms. On the one hand, updates forms are similar to input forms, because they have the same labels, text fields, and layout. On the other hand, their purpose and use are different. Update forms are used to modify or delete records from tables, whereas input forms are used to enter new records to tables. Also, the user has to specify an identifying attribute of the record to be updated in a preceding dialog box. Figure 7.6 shows the "Select Exercise" dialog box for the Operation Update Form.

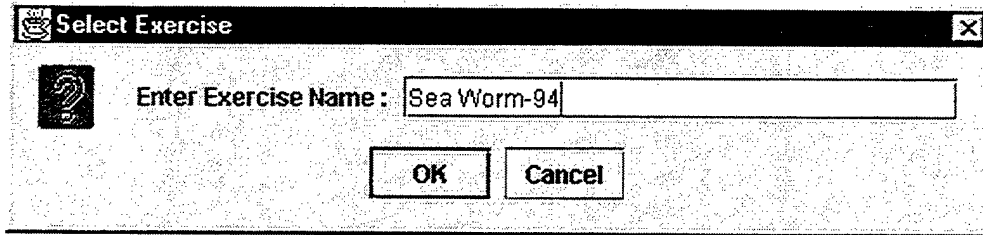


Figure 7.6: Select Exercise Dialog Box for Operation Update Form

When the user specifies the name of the exercise to be updated in the Exercise Name text field, the update form with “Delete Record”, “Update Record”, and “Cancel” options appear on the screen. Figure 7.7 shows the Operation Update Form.

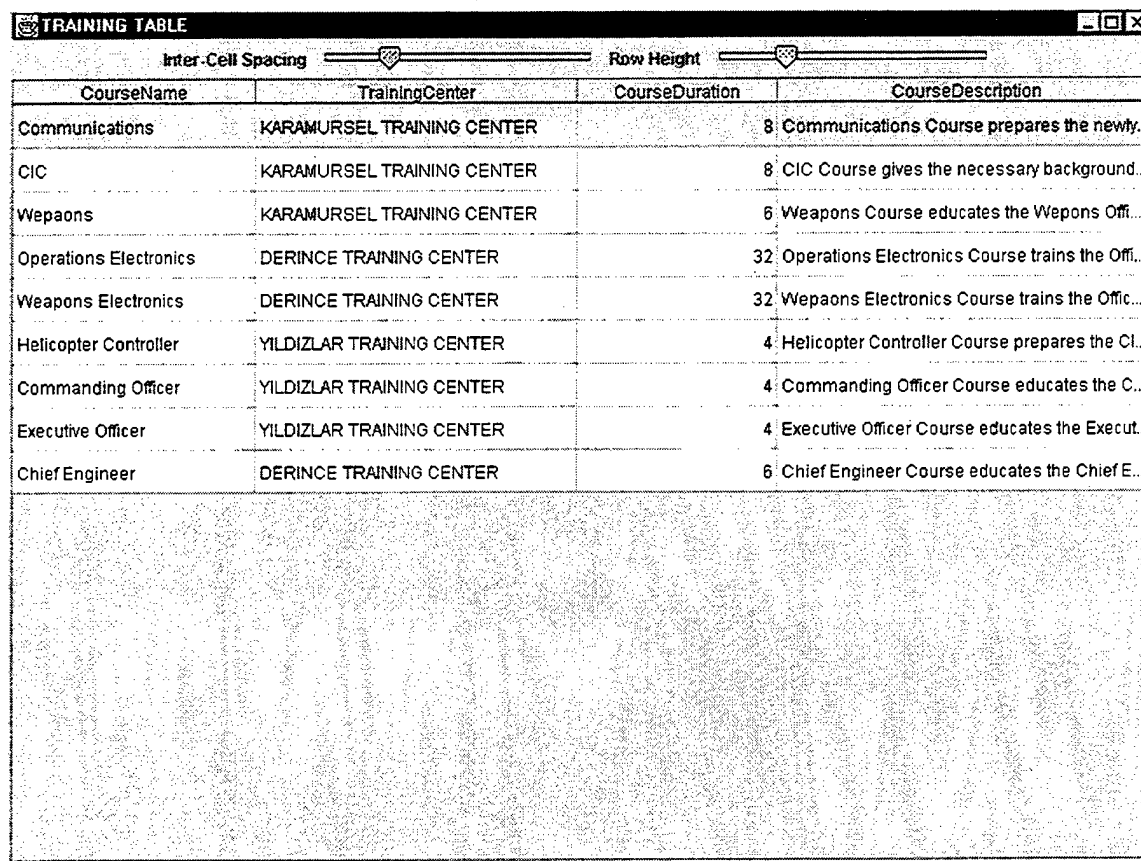
Exercise Name :	SEA WORM-94
Exercise Type :	FLEET EXERCISE
Start Date :	06/02/1994
End Date :	06/30/1994
Duration (Days) :	28
Place (Sea/Ocean):	Mediterranean Sea, Aegean Sea, Marmara Sea
Daytime Underway Hours :	482
Nighttime Underway Hours :	459
Helo Tail Number :	H41
Helo Flying Time (Hours) :	150
Number Of Dippings :	25
Total Dipping Time (Hours) :	65
Fuel Cost :	210000.0
Ammunition Cost :	80000.0
Amortization :	110000.0
Cost Of Exercise :	400000.0

ADD RECORD DELETE RECORD UPDATE RECORD CANCEL

Figure 7.7: Operation Update Form

3. Tables

Tables in the POET application program display all the records in a database table as a series of rows and columns, similar to the MS Access datasheet format. When the user clicks on the appropriate button, the corresponding table appears with its records. The following figure shows the Training Table.



CourseName	TrainingCenter	CourseDuration	CourseDescription
Communications	KARAMURSEL TRAINING CENTER	8	Communications Course prepares the newly...
CIC	KARAMURSEL TRAINING CENTER	8	CIC Course gives the necessary background...
Weapons	KARAMURSEL TRAINING CENTER	6	Weapons Course educates the Weapons Off...
Operations Electronics	DERINCE TRAINING CENTER	32	Operations Electronics Course trains the Off...
Weapons Electronics	DERINCE TRAINING CENTER	32	Weapons Electronics Course trains the Offic...
Helicopter Controller	YILDIZLAR TRAINING CENTER	4	Helicopter Controller Course prepares the Cl...
Commanding Officer	YILDIZLAR TRAINING CENTER	4	Commanding Officer Course educates the C...
Executive Officer	YILDIZLAR TRAINING CENTER	4	Executive Officer Course educates the Execut...
Chief Engineer	DERINCE TRAINING CENTER	6	Chief Engineer Course educates the Chief E...

Figure 7.8: Training Table

4. Reports

Reports are the main outputs that the POET database system generates for viewing the information in desired format. Reports can combine multiple tables to present the final output, which is created from different sets of data. This is accomplished by incorporating a query into the report design. POET application program provides the following reports, all of which contain data from two or more tables and all of which employ a static query.

- Overhaul Report
- Division Report
- Training Report
- Previous Assignment Report
- Foreign Language Report
- Event Report
- Port Visit Report
- Failure Report

The reports present the information in tabular format, as in Figure 7.9 for Port Visit Report.

PORT VISIT REPORT			
EXERCISE NAME	PORT NAME	START DATE	END DATE
DYNAMIC MIX-93	NAPOLI	10/10/1993	10/13/1993
SEA WORM-94	IZMIR	06/10/1994	06/13/1994
	ANTALYA	06/18/1994	06/20/1994
	MERSIN	06/23/1994	06/24/1994
FRIENDSHIP-97	VARNA	04/16/1997	04/18/1997
SEA WORM-96	CANAKKALE	06/09/1996	06/11/1996
	MARMARIS	06/15/1996	06/18/1996
	ANTALYA	06/22/1996	06/23/1996
DISTANT THUNDER-98	IZMIR	04/16/1998	04/18/1998
	ISTANBUL	04/22/1998	04/23/1998

Figure 7.9: Port Visit Report

5. Queries

Queries are used to extract information from the database. A query can select and define a group of records that fulfill a certain condition. The POET application program uses both dynamic and static queries to retrieve the information. In dynamic query case, the user can write the actual SQL statements that will return the desired records.

However, in static query case, the SQL code is already defined in the application program. The only thing that the user has to do is to specify the required selection condition, such as the last name in a query that shows the previous assignments of the person.

POET application program provides the following seven static queries, as well as the dynamic query.

- Courses-To-Take Query
- Courses-Taken Query
- Previous Assignments Query
- Foreign Languages Query
- Port Visits Query
- Events Query
- Failures Query

Similar to update forms, before executing the query, the user has to specify a selection condition in a dialog box. Upon entering the identifying criteria, the result of the query is presented in a datasheet format, which consists of a number of rows and columns. As an example, the dialog box and the result of the Exercise/Event Query are shown in Figure 7.10 and Figure 7.11, respectively.

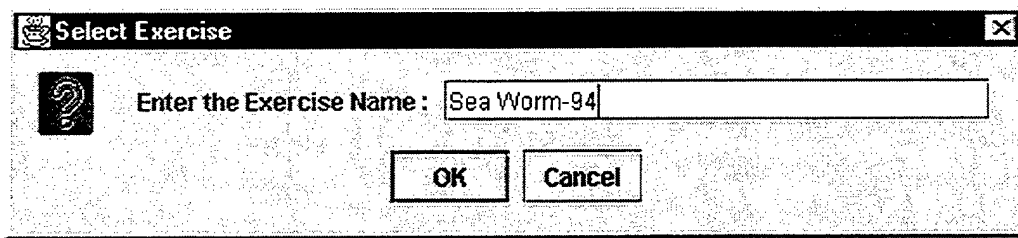


Figure 7.10: Select Exercise Dialog Box

EXERCISE/EVENT QUERY			
Inter-Cell Spacing		Row Height	
ExerciseName	EventName	EventType	EventDuration
SEA WORM-94	SURFEX-310	ANTISURFACE WARFARE	22
SEA WORM-94	SURFEX-316	ANTISURFACE WARFARE	11
SEA WORM-94	CASEX C-5	ANTISUBMARINE WARFARE	19
SEA WORM-94	EWX-320	ANTI-AIR WARFARE	10
SEA WORM-94	NAVCOMEX-408	COMMUNICATIONS	5
SEA WORM-94	NAVCOMEX-605	COMMUNICATIONS	4

Figure 7.11: Exercise/Event Query

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. SYSTEMS IMPLEMENTATION AND SUPPORT

The quality of an information system depends on its design, development, testing, and implementation. One aspect of systems quality is its reliability. A system is reliable if it does not produce dangerous or costly failures when used in a reasonable manner. An additional aspect of quality assurance is avoiding the need for enhancement on the one hand developing software that is maintainable on the other. The need for maintenance is high and impedes new developments. Maintenance and quality assurance needs are also better met when a structured development and documentation tool is used.

Quality assurance also includes testing to ensure that the system performs properly and meets its requirements. The purpose of testing is to find errors, not to prove correctness.

Implementation includes all those activities that take place to convert from the old system to the new. The new system may be totally new, replacing an existing manual or automated system, or it may be a major modification to an existing system. In either case, proper implementation is essential to provide a reliable system to meet organization requirements.

This chapter discusses the five aspects of systems implementation and support; including maintenance, quality assurance, reliability, training, and conversion.

A. SYSTEMS MAINTENANCE

When systems are installed, they generally are used for long periods of time. The average life of a system is four to six years, with the oldest applications often in use for over 10 years. However, this period of use brings with it the need to continually maintain the system. Because of the use a system receives after it is fully implemented, analysts must take precautions to ensure that the need for maintenance is controlled through design and testing and the ability to perform it is provided through proper design practices.

Many private, university, and government studies have been conducted to learn about maintenance requirements for information systems. The studies have generally concluded that: [Ref. 13]

- From 60 to 90 percent of the overall cost of software during the life of a system is spent on maintenance.
- Often maintenance is not done very efficiently. In documented cases, the cost of maintenance when measured on a per instruction basis is more than 50 times the cost of developing it in the first place.
- Software demand is growing at a faster rate than supply. Many programmers are spending more time on systems maintenance than on new development. Studies have documented that in some sites, two-thirds of the programmers are spending their time on the maintenance of software.

Information systems and the organizations they serve are in a constant state of flux. Therefore, the maintenance of systems also involves adaptations of earlier versions of the software. Approximately one-fifth of all maintenance is performed to accommodate changes in reports, files, and data.

The greatest amount of maintenance work is for user enhancement, improved documentation, or recording systems components for greater efficiency. Sixty percent of all maintenance is for this purpose. Yet, many of the tasks in this category can be avoided if systems' engineering is carried out properly. The design practices followed for software dramatically affect the maintainability of a system: good design practices produce a product that can be maintained.

The keys to reducing the need for maintenance, while making it possible to do essential tasks more efficiently, are: [Ref. 13]

- More accurately defining user requirements during systems development
- Assembling better systems documentation
- Using more effective methods for designing processing logic and communicating it to project team members
- Making better use of existing tools and techniques
- Managing the systems engineering process effectively

B. QUALITY ASSURANCE

Quality assurance is the review of software products and related documentation for completeness, correctness, reliability and maintainability. And it, of course, includes assurance that the system meets the specifications and requirements for its intended use and performance.

There are four levels of quality assurance: testing, verification, validation, and certification.

1. Testing

Systems testing is an expensive, but critical, process that may take as much as 50 percent of the budget for program development. The common view of testing is that it is performed to prove that there are no errors in a program. However, as indicated earlier, this is virtually impossible since analysts cannot prove that software is free and clear of errors.

Therefore, the most useful and practical approach is with the understanding that *testing* is the process of executing a program with the explicit intention of finding errors, that is, making the program fail. The tester, who may be an analyst, programmer, or specialist trained in software testing, is actually trying to make the program fail. A successful test, then, is one that finds an error.

2. Verification

Like testing, *verification* is also intended to find errors. It is performed by executing a program in a simulated environment. When commercial systems are

developed with the explicit intention of distributing them to dealers for sale or marketing them through offices, they first go through verification, which is sometimes, called *alpha testing*.

3. Validation

Validation refers to the process of using software in a live environment in order to find errors. The feedback from the validation phase generally produces changes in the software to deal with errors and failures that are uncovered. Then a set of user sites is selected that put the system into use on a live basis. These *beta test* sites use the system in day-to-day activities; they process live transactions, and produce normal system output. The system is live in every sense of the word, except that the users are aware they are using a system that can fail. Validation may continue for several months. During the course of validating the system, failure may occur and the software will be changed.

4. Certification

Software *certification* is an endorsement of the correctness of the program. This is an issue that is rising in importance for information systems applications. To certify the software, the agency appoints a team of specialists who carefully examine the documentation for the system to determine what the vendor claims the system does and how it is accomplished. Then they test the software against those claims. If no serious discrepancies or failures are encountered, they will certify that the software does what the documentation claims. They do not, however, certify that the software is the right package for a certain organization. [Ref. 13]

5. Testing Strategies

It is already indicated that the philosophy behind testing is to find errors. Test cases are devised with this purpose in mind. A *test case* is a set of data that the system will process as normal input. However, the data is created with the intent of determining whether the system will process it correctly. Each test case is designed with the intent of finding errors in the way the system will process it.

There are two general strategies for testing software. This section examines both, the strategies of code testing and specification testing.

a. Code Testing

The *code-testing* strategy examines the logic of the program. To follow this testing method, the analyst develops test cases that result in executing every instruction in the program or module. That is, every path through the program is tested. A *path* is a specific combination of conditions. [Ref. 13]

On the surface, code testing seems to be an ideal method for testing software. The rationale that all software errors can be uncovered by checking every path in a program is faulty. First of all, in even moderately large programs of the size used in typical business situations, it is virtually impossible to do exhaustive testing of this nature. Financial and time limitations alone will usually preclude executing every path through a program since there may be several thousand.

However, even if code testing can be performed in its entirety, it does not guarantee against software failures. This testing strategy does not indicate whether the code meets its specifications nor does it determine whether all aspects are even

implemented. Code testing also does not check the range of data that the program will accept even though when software failures occur in actual use, it is frequently because users submitted data outside of expected ranges.

b. *Specification Testing*

To perform *specification testing*, the analyst examines the specifications stating what the program should do and how it should perform under various conditions. Then test cases are developed for each condition or combination of conditions and submitted for processing. By examining the results, the analyst can determine whether the program performs according to its specified requirements. [Ref. 13]

This strategy treats the program like a black box. That is, the analyst does not look into the program to study the code and is not concerned about whether every instruction or path through the program is tested.

Neither testing strategy is ideal. However, specification testing is a better strategy since it focuses on the way software is expected to be used.

C. TRAINING

Even well designed and technically elegant systems succeed or fail because of the way they are operated and used. Therefore, the quality of training the personnel involved with the system in various capacities helps or hinders, and may even prevent, the successful implementation of an information system. Those who will be associated with or affected by the system must know in detail what their roles will be, how they may use the system, and what the system will or will not do.

User training must ensure that they are able to handle all possible operations, both routine and extraordinary. Training involves familiarization with run procedures, that is, working through the sequence of activities needed to use a new system on an ongoing basis. The operators should also be instructed in the common malfunctions that can occur, how to recognize them, and what steps to take when they arise.

User training must also instruct individuals in troubleshooting the system, determining whether a problem arising is caused by the equipment or software or by something they have done in using the system. Including a troubleshooting guide in systems documentation will provide a useful reference long after the training period is over. There is nothing more frustrating than working with a system, encountering a problem, and not being able to determine whether it is your fault or a problem with the system itself. The place to prevent this frustration is during training.

As the above discussion demonstrates, there are two aspects to user training: familiarization with the processing system itself (that is, the equipment used for data entry and processing) and training in using the application (that is, the software that accepts the data, processes it, and produces the results). Weaknesses in either aspect of training are likely to lead to awkward situations that produce user frustration, errors, or both. Good documentation, although essential, is not a substitute for training. There is no substitute for hands-on operation of the system while a person is learning how to use the program.

D. CONVERSION

Conversion is the process of changing from the old system to the new one. There are four methods of handling a systems conversion. Each method should be considered in light of the opportunities it offers and problems that it may cause. However, some situations force one method to be used over others, even though other methods may be more beneficial. In general, systems conversion should be accomplished as quickly as possible. Long conversion periods increase the possible frustration and difficulty of the task for all people involved. [Ref. 13]

1. Parallel Systems

The most secure method of converting from an old to new system is to run both systems *in parallel*. That is, users continue to operate the old system in the accustomed manner but they also begin using the new system. This method is the safest conversion approach since it guarantees that, should problems arise in using the new system, such as errors in processing or inability to handle certain types of transactions, the organization can still fall back to the old system without loss of time, revenue, or service.

The disadvantages of the parallel systems approach are significant. First of all, the system costs double since there are now two sets of systems costs. In some instances, it is necessary to hire temporary personnel to assist in operating both systems in parallel. Second, the fact that users know they can fall back to the old ways may be a disadvantage if there is potential resistance to the change or if users prefer the old system. [Ref. 13]

All in all, the parallel method of systems conversion offers the most secure implementation plan if things go wrong, but the costs and risks to a fair trial cannot be overlooked.

2. Direct Conversion

The *direct conversion* method converts from the old to the new system abruptly, sometimes over a weekend or even overnight. The old system is used until a planned conversion day. Then it is replaced by the new system. There are no parallel activities.

If the management must make the change and wants to ensure that the new system fully replaces the old one so that users do not rely on the previous methods, direct conversion will accomplish this goal. Psychologically, it forces all users to make the new system work; they do not have any other method to fall back on. The advantage of not having a fallback system can turn into a disadvantage if serious problems with the new system arise. In some instances, organizations even stop operations when problems arise so that difficulties can be corrected. [Ref. 13]

Direct conversion requires careful advanced planning. Training sessions must be scheduled and maintained. The installation of all equipment must be on time, with ample days allowed in the schedule to correct any difficulties that occur. Any site preparation must be complete before the conversion can be done.

3. Pilot Approach

When new systems also involve new techniques or drastic changes in organization performance, the *pilot* approach is often preferred. In this method, a working version of the system is implemented in one part of the organization, such as a single work area or department. The users in this area typically know that they are piloting a new system and changes may be made to improve the system.

When the system is deemed complete, it is installed throughout the organization, either all at once (direct conversion) or gradually (phase-in). [Ref. 13]

This approach has the advantage of providing a sound proving ground before full implementation. However, if the implementation is not properly handled, users may develop the impression that the system continues to have problems and cannot be relied on.

4. Phase-In Method

The *phase-in* method is used when it is not possible to install a new system throughout an organization all at once. The conversion of files, training of personnel, or arrival of equipment may force the staging of the implementation over a period of time, ranging from weeks to months. Some users will begin to take advantage of the new system before others. [Ref. 13]

Long phase-in periods create difficulties for analysts, whether the conversions go well or not. If the system is working well, early users will communicate their enthusiasm to others who are waiting for implementation. In fact, enthusiasm may reach such a high level that when a group of users does finally receive the system, there is a letdown. Later,

when conversion occurs, the staff finds out that the system, even though working properly, does not do the processing instantly.

On the other hand, if there are problems early in the phased implementation, word of difficulties will spread also. Then the users may expect difficulties when they are converted and react negatively to the smallest mistakes, even their own.

E. SYSTEMS RELIABILITY

A *reliable system* is one that does not produce dangerous or costly failures when used in a reasonable manner, that is, in a manner that a typical user expects is normal. This definition recognizes that systems may not always be used in the ways that designers expect.

There are two levels of reliability. The first is that the system meeting the right requirements. If it is expected to have specific security features or controls, but the design fails to specify them, then the system is not reliable. Reliability at the design level is only possible if a through and effective determination of systems requirements was performed by the analyst. A careful and through systems study is needed to satisfy this aspect of reliability.

The second level of systems reliability is the actual working of the system delivered to the user. At this level, systems reliability is interwoven with software engineering and development. [Ref. 13]

IX. ANALYSIS OF POET DATABASE SYSTEM

This chapter analyzes the use, benefits and installation of the POET Database System that will support the administrative activities on the Turkish Navy frigates by storing, processing, and accessing the personnel, operations, equipment, material and training data.

The chapter is divided into four sections. In the first section, the chapter gives a brief introduction about the current situation of information processing in the Turkish Navy frigates and then compares the file-processing systems with the database processing systems. The second section provides an analysis of the benefits of the system from managerial, manpower, and technical aspects. The third section analyzes the system implementation and installation issues by explaining the training and conversion phases in detail. In the last section, the impact of the computer technology in the organizations is examined.

A. CURRENT SITUATION

Today, the frigates are the most effective, powerful, and capable vessels among the warships. The modern frigates are designed as multi-purpose combatants, which can be used in anti-air, anti-surface, and anti-submarine warfare. They can serve as both offensive and defensive vehicles according to the needs of the circumstances.

As it is the case in most navies, the frigates constitute the main force of the Turkish Fleet. The Turkish Navy frigates have a challenging mission, which encompasses tactical, operational, and administrative tasks. Because of their powerful weapons and

maneuvering capabilities, the frigates are also the most active warships in the Turkish Navy. They participate in lots of operations, exercises, and maneuvers while conducting a number of deployments. This intense tempo causes a heavy burden of administrative and bureaucratic tasks. There is a large volume of reports, messages, and documents that are required either for the submission to the higher command or for the ship's internal use. The documents to be generated may be periodic reports, prepared daily, weekly, monthly, bimonthly, and annually, or they may be ad hoc reports that may be requested anytime.

In the Turkish Navy frigates, it is a time consuming process to prepare some documents, because the information needed is not stored in a single and central database management system. The organization of the frigates consists of six departments, which are Operations, Engineering, Electronics, Navigation, Weapons, and Supply Departments. Each department in the ship keeps its data in different formats and environments, such as Microsoft Word, Microsoft Excel, Word Perfect, Frame Maker, or other special application programs. There is neither a standard format nor a software program to store, manipulate, and access the data. When it is required to generate a report, which will include information from two or more departments, a person from the administrative office has to collect that data from the departments manually.

This technique, which is known as File-Processing System, has many drawbacks compared to the Database Processing Systems.

B. FILE-PROCESSING SYSTEMS

The file-processing systems, which predated the use of database processing systems, are a great improvement over manual record keeping systems, however they have the following limitations. [Ref. 3]

1. Data Redundancy

The first drawback of file-processing systems is *uncontrolled redundancy of data* and *data duplication*. Since every department keeps the data in a different environment and different file, generally the same information is stored in more than one department's files. For instance, both the Operations and the Navigation departments store the data about the exercises and navigation times. This results in the same data being stored in separate files that must exist for each different system.

2. Data Inconsistency

Although the duplicate data wastes personnel time and file space, that is not the most serious problem; rather, the biggest problem with the duplicate data concerns *data consistency* and *data integrity*. Poor data integrity can often be seen in file-processing systems, because it is very difficult to keep the redundant data consistent. The greater the degree of redundant data, the more difficult it becomes to insure that the data is accurate and timely. For example, if the start and end times of an exercise change, then each file containing this data must be updated, but the danger is that all of the files might not be updated, causing discrepancies among them.

Therefore, a report from the Operations Department may disagree with a report from the Navigation Department. When the retrieved information are inconsistent, the credibility and the reliability of the stored data comes into question.

3. Limited Sharing of Data

Another disadvantage of file-processing systems is the *limited sharing of data*. Because of their different formats and file structure incompatibilities, the data files can not be readily combined or compared. This limitation restricts the extent to which applications are able to share each other's data. The difference in the definitions of data among the various files will result in difficulty for users to cross-reference data elements in other files. Suppose that it is necessary to compare the electronic equipment inventory of the Supply Department to that of the Electronics Department. Since they store their inventories in different environments and different file structures, this would be a burdensome and time-consuming process.

4. Program/Data Dependency

In file-processing systems, *application programs depend on the file formats*, because the physical formats of files and records are usually part of the application code. Therefore, individual data structures were developed for specific applications, and each program within each system has been designed to process the records and data items of its corresponding files. The problem with this arrangement is that when changes are made in the file formats and data structures, such as adding a new record type, the application programs also must be changed.

This makes it difficult to introduce changes and results in high maintenance programming costs to cope with the inevitability of such changes.

5. Inflexibility of Information

Inflexibility of available information is another drawback of file-processing systems. Files tend to isolate the information and restrict what can be retrieved. It is not possible to get the information in a modified format, which can be quite useful for some administrative purposes. The effectiveness of an organization's data resources require flexibility, accuracy, the ability to support adequate response levels to inquiries, and a large degree of data sharing across various applications.

6. Data Isolation

In file-processing systems, *data are separated and isolated*. Sometimes it may be desirable to combine two or more files into a single file to retrieve the required information, but with file processing, this is a very exhausting process. Generally, the information that requires the combination of two or more files is gathered by a person instead of the computer.

7. Difficulty in Representing Data

It is difficult to represent file-processing data in a form that seems natural to users. Users want to see all related data in a single document, but in order to show the data in this way, several different files need to be combined and the result is to be presented together. For example, if a higher command wants a report from the ship after

each deployment that includes information about conducted exercises and training, navigational and operational data, and the amount of ammunition and fuel used during the deployment. In order to prepare such a report, it is necessary to combine all of the departments' files together.

8. Difficulty in Information Resource Management

Information resource management is more difficult in file-processing systems. Especially, data security is very difficult to achieve, because multiple files in separate places cause more vulnerability. On the other hand, it is a big problem to enforce some ship-wide and navy-wide standards on data storage systems.

C. DATABASE PROCESSING SYSTEMS

The database concept not only offers distinct advantages from the conventional file processing approach, but also results in added value to an organization in what it can realize from its data resources. A database system can significantly enhance the quality of information, provide the basis for increased efficiency in programming, and introduce tools for the effective management of information at all levels of an organization. Database processing systems overcome the limitations of file-processing systems, because file-processing programs store the data in different files and directly access the files of stored data, while database-processing programs keep the data in a single database and invoke the Database Management System (DBMS) to access the stored data. DBMS is a set of programs used to define, administer, and process the database and its applications. [Ref. 3]

In a database system, all of the data is stored in a single facility, called the *database*. When this approach is applied to the Turkish Navy frigates, the departments in the ship will no longer have separate data files; on the contrary, there will be a centralized database, which will store information about personnel, operations, equipment, training, and logistics, and an application program with a graphical user interface that will allow the users to access the data in a windows-based environment. The application program retrieves the required data by invoking the DBMS, which accesses and manipulates the database. The following section explains the advantages of database processing systems over the file processing systems in detail.

1. Minimum Data Redundancy

With database processing, *the data redundancy is minimal* as a result of the centralization and the increased capability of data sharing. It is no longer necessary to store the same information for different applications, because all programs have access to the same data. The decrease in data redundancy results in greater data accuracy. Since the data is stored in only one place, data integrity problems are less common. There is less opportunity for inconsistency among multiple copies of the same data item, because whenever the data is modified, only one update is sufficient. By integrating data into a common location, standard program definitions of data can be established to satisfy multiple uses.

2. Improved Data Sharing

Sharing of data is improved with the database processing approach. Data are organized into a single, centralized database that allows combination of files and separate views of the data. With database processing, it is possible to produce more information from a given amount of data, because the tables/files are related to each other via common data fields and they can be easily combined.

3. Increased Data Availability

Improved data sharing, searching versatility, and multiple views of data present to programmers and users a data resource that can satisfy their demands for information. The database, through ease of access, can increase *data availability*. Throughout the organization, data will no longer be thought of as something that is the exclusive province of the computer and the people who help run it.

4. Cost Reduction

A database environment increases the potential for the *reduction of cost*, especially in one significant area: maintenance programming. Maintenance programming costs are minimized through data independence by eliminating re-programming due to changes in physical and logical data definitions.

5. Flexibility in Data Access

Database processing systems provide *more flexibility in data access and retrieval*. Ad hoc reports as well as routine ones can be quickly prepared with this approach,

because the DBMS supports programmed and unprogrammed queries via Structured Query Language (SQL). For example, it takes a few minutes to generate the deployment report, since the tables can be joined together and the DBMS can be used to make the appropriate query.

6. Advanced Security and Integrity

Another significant advantage of database processing systems is the *advanced security, privacy, and integrity controls*. Since the data is stored in a centralized database, security issues can be more effectively supported by giving accounts and passwords to the authorized users. Also, navy-wide standards can be enforced in this system. An integrated database will allow an organization to establish stringent access controls for specific entities and their data items. For example, if personnel evaluation information is not to be divulged, security provisions such as passwords can be assigned to segments of the database. User or program access to this data, therefore, would only be allowed if the proper authorization (e.g., use of a password) was established.

7. Program/Data Independence

Database processing reduces the dependency of application programs on file formats, because a database can insulate an application program from changes made to the structure of the data it uses. All record formats, along with the data, are stored in the database itself, and they are accessed by the DBMS, not by the application programs. *Program/data independence* minimizes the impact of data format changes on application programs and provides great flexibility and efficiency. The data processing environment

is never static and the need for new information is ever present. By protecting existing programs from these requirements, it is possible to satisfy these demands without investing the people resources and money to change existing programs. In a conventional file environment, changes to data structures almost always result in changes to those application programs.

8. Dynamic Structure

Data independence also provides for the introduction of new technology and processing techniques without the necessity of constant re-programming. A database can be transferred to new storage devices or re-organized in ways that will enhance access response time while leaving application programs unaffected.

D. BENEFITS OF THE POET DATABASE SYSTEM

1. Technical Aspect

The purpose of the database system is to store information about personnel, operations, equipment, and training in a centralized database, to generate standard reports, to provide ad hoc queries, and to support the administrative activities onboard the Turkish Navy frigates. The database program will minimize the data redundancy and increase the availability, accuracy, consistency, security, and integrity of data.

The database management system can be implemented cost effectively by using Microsoft Access, which is a popular and economical Commercial Of The Shelf (COTS) product. The application program that will provide the graphical user interface between

the user and the database management system can be developed with Java, which is a very powerful and efficient yet free object-oriented programming language. The system can be operated on a Personal Computer (PC), preferably the computer in the administrative office. The administrative office personnel can be granted user accounts and passwords, which will provide the control and security for the classified data.

The system can be designed with a user friendly graphical user interface, and therefore the users of the system will be able to learn the program without the need of an extreme training and advanced computer experience. As a result, this database application program will provide the Turkish Navy with an affordable and efficient system that will support administrative activities on the frigates and enforce the navy-wide standards.

2. Manpower Aspect

In the current situation, at least one person is assigned to input, manipulate, and analyze the data in each department. Besides, there is an administrative office that is responsible for generating ship-wide reports and documents and for conducting the common managerial tasks. There are four crewmembers stationed in this office, a petty officer and three enlisted personnel. The Communications Officer, who is responsible for all incoming and outgoing documents and messages, is in charge of the administrative office. The main function of this office is to keep track of incoming documents and to prepare the required documents and reports.

Since the ship lacks a centralized database system to store and access the data, the administrative office personnel spend almost one-half of their time to gather the necessary information from the departments. Counting the personnel responsible for data storage and retrieval in six departments, there are about twelve crewmembers who strive to achieve the same task: keep the data in a file, process it, and access the information needed. This task is the main functionality of a database system, which provides much more capabilities.

The use of a database system would greatly reduce the productive power loss and work hours spent on administrative tasks that are instrumental in accomplishing the Turkish navy frigates' principal tasks. In order to run such a database system effectively, the administrative office personnel would be sufficient, because with this system they could save a lot of time and would not have to collect the information manually. This means that by using this database system, the other six crewmembers may be employed more efficiently in other tasks, which results in considerable saving in personnel power.

3. Decision Making Aspect

The Commanding Officer (CO), the Executive Officer (XO) and the Department Heads may need to make decisions in a short period of time. As managers of the ship, they want to analyze the situation by examining the available information that will help them make their decisions. The decision making process is based on two decision elements – the amount of information used in making a decision and the number of alternatives considered [Ref. 14]. With respect to information use, the person in the decision-making role, wants as much relevant information as possible before reaching a

decision. Nobody, it seems, wants to make a decision without having the necessary information, because it increases the possibility of choosing the wrong alternative.

If the CO needs information to make a decision, having personnel collect the data in order to propose a suggestion is a time-consuming process and it might produce inconsistent and unreliable results. However, the use of a database system can help the decision making process in the ship by providing accurate and timely information.

E. INSTALLATION OF THE POET DATABASE SYSTEM

1. Training

The training for the database system should be designed so that every user of the system knows the system's features and functions and has knowledge about how to use the system. The success of any information system depends on the skills of the operators. In the Turkish Navy frigates, the main users of the database system will be the personnel working in the ship's administrative office, who are familiar with the computers and procedures on the ship and who only need to be trained on how the new system operates. The Communications Officer leads the administrative office, whose personnel normally consist of one petty officer and three enlisted personnel.

An important factor that can affect the success of the system and the training phase is the design and implementation of the system. The database system and the application program should be designed so that it is easy-to-use and does not require the operators to have any advanced computer science knowledge. However, matching basic

human characteristics and skills with a job's requirements is essential, especially when an automated system is to replace a manual one.

The training should make the users of the system familiar with the system's interface as well as its functions and capabilities. Once the frigates have their administrative office personnel trained, educating the other potential users on how to use the system can be handled within the ship.

2. Conversion

The conversion from the old system to the new one is one of the most important phases of the installation from a managerial perspective. This normally because human nature is inclined to resist changes, especially if they are not ready or prepared for these changes. Factors such as organizational structure, human resources, and cultural climate all come into play. Managers sometimes have to restructure the organization chart when a computerized system is implemented. Human resources are often reallocated to and from the new system in order to increase the efficiency of the new system. Conflicts among the users and the personnel who don't believe in the benefits of the new system should be expected. Furthermore, some people view training programs as a threat, because they believe that evaluations made at the end of the training may be a negative factor in their career.

When the new database system is built on the Turkish Navy frigates, the crewmembers assigned to data storage tasks in the departments can be assigned to new jobs.

Once the administrative office personnel are trained and gained the skills to use the new system, they can handle the administrative activities by themselves without any extra personnel support. [Ref. 15]

The four conversion strategies that can be used to install a new system in an organization are already explained in Chapter II. For the POET Database Application program, the Pilot Approach is proposed as the most appropriate strategy for the Turkish Navy frigates. Two frigates must be selected as the pilot units in order to get sufficient feedback. The conversion method to be applied to the pilot ships should be the Parallel Systems approach for the following reasons.

- Risk is significantly minimized;
- Testing the system on two ships over a period of time will provide sufficient information to evaluate the system before complete implementation on all ships;
- Manpower need is reasonable;
- Surfaced problems can be worked out by the personnel of both ships;
- Time to shift is predicted to be two months, which is a reasonable interval for checking monthly reports and for reassigning personnel from/to the ship.

As soon as the system operates efficiently on the pilot units, a decision for full implementation on all ships can be made.

3. Integration

Integration of the converted files and applications into the database environment has to be planned. For some period of time, there will be two sets of applications that will run concurrently. Personnel must be trained for the new environment. The organization can't be closed for the conversion and integration phases. The file environment is most probably the production environment, and the data base environment will be the test environment. The updates performed to the production files have to be transferred to the test environment. It is important that the two systems represent the same information at all times.

F. ASSESSING THE IMPACTS OF COMPUTER TECHNOLOGY IN ORGANIZATIONS

Computer is often identified as the key device in the third revolution of humankind. There are two powerful and contrary images widely linked with the use of computer technology in organizations. In one view, the computer is the great problem solver, producing important gains in the efficiency and effectiveness of personnel in their work. In the contrasting view, computer is a problem generator – an expensive and disruptive technology that has often failed to match its promises in many of the actual tasks to which it has been applied. [Ref. 16]

There are some reasons why an organization might adopt computer technology. The organization might use computers in order to symbolize its commitment to modern management practices or to advanced technology. Or the organization might want to

indicate that its decisions and actions are guided by information systems, which can process large amounts of data, rather than relying upon manual methods of information storage and retrieval. Actually, the major reasons for computer utilization usually involve expectations that computers will generate real benefits in information processing, and ultimately in organizational performance.

With more than 200 crewmembers as the personnel and the CO and the XO as the managers, the Turkish Navy frigates are no different than the other organizations in the business world. Until now, the use of computer technology in the Turkish Navy frigates have increased the technical and managerial capabilities of the ships, while helping the managers in their decision making process and providing more time for the necessary exercises that must be conducted by the personnel. Actually, the phenomenal expansion in the use of computer technology is a significant proof that computers have generally produced successful results in the organizations.

Among the benefits that might be anticipated from the use of computers in organizations, those that improve the information environment are perhaps the most obvious. The extensive information handling capabilities of computers are often used in the field of data processing and database applications. The database system that would be installed on the Turkish Navy frigates is expected to enhance the ship's administrative capabilities to save personnel power and time, to improve data accuracy, consistency and timeliness, and to provide the Turkish Navy with an affordable and efficient system.

The potential benefits of such a database system can be listed as follows:

- the *high speed* with which information can be obtained
- the *ease of access* to information
- the *availability* of new information
- the *timeliness* of the information
- the *accuracy* and the *consistency* of information
- the *savings* in the personnel employment
- the *improvements* in the decision making process

G. CONCLUSION

From personal experience, the author served as a Communications Officer on TCG YILDIRIM (F-243), one of the newest frigates of the Turkish Navy, for more than three years. As the Communications Officer of the ship, the position included leadership of the administrative office, in which one petty officer and two enlisted personnel were stationed. When the ship is at the port, 80% of the time was spent on administrative tasks, which mostly consisted of preparing reports and documents, and collecting the information needed to generate these documents.

While doing/managing these activities, it was noticed by the personnel in the administrative office as they felt the need for an information/database system that could be used to store, process and access required the data.

By using such a database system that automates most of the manual tasks, it could possibly save valuable time and personnel power while increasing the work volume and efficiency of the administrative office.

As a result of having such an experience, it was desired that Turkish Navy frigates had had this database application program to produce effective, quality reports that could be analyzed with accuracy toward continual improvement in ship operations.

THIS PAGE INTENTIONALLY LEFT BLANK

X. CONCLUSIONS

A. SYNOPSIS

This thesis presented the design, development, implementation, and analysis of the Personnel, Operations, Equipment, and Training (POET) Database and Application Program for the Turkish Navy frigates on a standalone computer. The POET database system will provide the Turkish Navy ships with an automated system to perform their primary administrative functions. POET will support this mission by keeping track of all the personnel, operations, equipment, and training records, maintaining them, producing standard reports and providing the command with ad hoc information.

Besides implementing a database and an application program, this thesis has also specified the current situation and the need for such a database application program onboard Turkish Navy frigates. The main goal of developing the POET database system is to release manpower to perform other duties and to reduce the productive power loss by increasing the availability, accuracy, efficiency, and consistency of the data needed in administrative activities. This program is expected to eliminate most of the current problems and to result in considerable savings of personnel power and time while providing the required information to the command quickly.

After examining the current methods used to store and retrieve information in the Turkish Navy frigates, requirements collection and analysis phase was performed to determine the expected functionality of the POET database system. This is the most important step of the entire database design process, because most subsequent design decisions are based on this step.

The major tasks of this phase are to specify the data requirements and to determine the functional requirements.

Once all the requirements have been collected and analyzed, a conceptual schema was created for the database by using Semantic Object Modeling technique to capture the user requirements. In order to build an effective database and related applications, a data model that captures the users' perceptions closely is of great importance. The data model should identify the entities and their attributes to be stored in the database and should define their structure and the relationships among them.

After the data model is developed as semantic objects in the conceptual database design phase, these objects were transformed into an implementation data model. In this thesis, the relational model, which is the most common data model used in commercial DBMSs, was used as the implementation data model. Then, the POET database was implemented in Microsoft Access.

Java programming language and Java Database Connectivity (JDBC) application program interface was used as a tool for developing an application program, which will eliminate the need for a Database Management System environment.

Upon completion of the program and implementation, an evaluation and analysis the possible benefits and advantages that would be gained by using POET system from manpower, management, and technical perspectives was done.

It is hoped that this system, as an initial effort, will be the motivator for other efforts to develop new systems and benefit other branches of the Turkish Navy and as an inspiration to develop additional dedicated, focused systems.

B. FUTURE ENHANCEMENTS

The POET database system is developed in order to help a ship's personnel perform the administrative activities by implementing a centralized database. The system is designed as a single-ship system to be used on individual frigates. As a future enhancement, it can be expanded and redesigned so that it can be employed in Destroyer Division Commands and Fleet Command. In such a case, the personnel in Fleet Command will be able to store and access the required information about all of the ships.

A program that will provide all message and report formats in a text editor environment and that will produce the standard reports by integrating the format with the data from POET database may be developed. Combining the capabilities of these two programs, the command can save a lot of manpower by automating the report generation process.

Another research area is to make the reports and messages produced by the POET database system available in the military network. This facility will allow the ships to electronically transfer their reports and messages to the higher command efficiently and quickly while reducing the amount of paper work at the point of origin and receiving unit.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: SEMANTIC OBJECTS

SHIP

ID InternationalCallSign 1,1

ShipName 1,1

HullNumber 1,1

ShipClass 1,1

KeelLayingDate 1,1

LaunchDate 1,1

CommissionDate 1,1

ShipLength 0,1

ShipWidth 0,1

MastHeight 0,1

KeelDepth 0,1

Displacement 0,1

HomePort 1,1

SuperiorInCommand 1,1

Overhauls

ID OverhaulNumber 1,1

StartDate 1,1

EndDate 1,1

ShipyardsName 1,1

OverhaulDuration 1,1

0,N

SHIP

PlannedManning

PlannedOfficers 1.1

PlannedPettyOfficers 1.1

PlannedEnlisted 1.1

1.1

PresentManning

PresentOfficers 1.1

PresentPettyOfficers 1.1

PresentEnlisted 1.1

1.1

DEPARTMENT

ID DepartmentName 1.1

PlannedManning

PlannedOfficers 1.1

PlannedPettyOfficers 1.1

PlannedEnlisted 1.1

1.1

PresentManning

PresentOfficers 1.1

PresentPettyOfficers 1.1

PresentEnlisted 1.1

1.1

DIVISION

0.N

PERSONNEL

1.N

DIVISION

ID DivisionName 1.1

PlannedManning

PlannedOfficers 1.1

PlannedPettyOfficers 1.1

PlannedEnlisted 1.1

1.1

PresentManning

PresentOfficers 1.1

PresentPettyOfficers 1.1

PresentEnlisted 1.1

1.1

DEPARTMENT

1.1

PERSONNEL

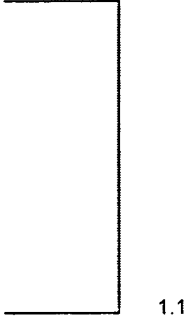
1.N

PERSONNEL

ID MilitaryID 1,1
FirstName 1,1
LastName 1,1
Rank 1,1
Rating 1,1
DateOfBirth 1,1
PlaceOfBirth 1,1
FatherName 0,1
MotherName 0,1
ActiveDutyServiceDate 0,1
DateOfRank 0,1
Gender 1,1
MaritalStatus 1,1
SpouseName 0,1
NumberOfChildren 0,1

Address

Street 1,1
City 1,1
State 1,1
ZipCode 1,1



PERSONNEL

PhoneNumber 1.1

Speciality 1.1

Education 1.1

CurrentAssignment 1.1

StartDate 1.1

CabinNumber 1.1

CabinPhone 1.1

PreviousAssignments

ID AssignmentNumber 1.1

Station 1.1

Position 1.1

Duration 1.1

0.N

Courses_Taken

ID TRAINING 1.1

StartDate 1.1

EndDate 1.1

Grade 1.1

0.N

Courses_To_Take

ID TRAINING 1.1

0.N

Foreign_Languages

ID Language 1.1

Degree 1.1

0.N

TRAINING

ID CourseName 1,1

TrainingCenter 1,1

Duration 1,1

CourseDescription 0,1

PERSONNEL

0..N

PERSONNEL

0..N

EQUIPMENT

ID SerialNumber 1.1

StockNumber 1.1

EquipmentName 1.1

EquipmentType 1.1

Manufacturer 1.1

Model 1.1

ProductionDate 1.1

Location 0.1

Runtime 1.1

Failures

ID FailureNumber 1.1

Description 1.1

Diagnosis 1.1

FailureDate 1.1

FailureDuration 1.1

0.N

OPERATION

ID ExerciseName 1,1

ExerciseType 1,1

StartDate 1,1

EndDate 1,1

Duration 1,1

Place 1,1

Events

ID EventName 1,1

EventType 1,1

EventDuration 1,1

NumberOfEvents 1,1

1.N

Port_Visits

ID PortName 1,1

VisitStartDate 1,1

VisitEndDate 1,1

VisitDuration 1,1

0.N

Underway_Durations

DaytimeHours 1,1

NighttimeHours 1,1

1,1

OPERATION

Cost_Of_Exercise

FuelCost 1.1

AmmunitionCost 1.1

Amortization 1.1

CostOfExercise 1.1

1.1

Helicopter

ID HelotailNumber 1.1

FlyingDuration 1.1

NumberOfDippings 1.1

DippingDuration 1.1

0.1

APPENDIX B: DOMAIN SPECIFICATIONS

A. SHIP OBJECT

Domain Name	Type	Semantic Description	Physical Description
International CallSign	Simple Attribute	International Call Sign of the Ship	Text 4; Capital Letters
ShipName	Simple Attribute	Name of the Ship	Text 30
HullNumber	Simple Attribute	Hull Number of the Ship	Text 4; One capital letter and three digits
ShipClass	Simple Attribute	Class of the Ship	Text 50
KeelLayingDate	Simple Attribute	Keel Laying Date	Text 10; format 00/00/0000
LaunchDate	Simple Attribute	Launch Date	Text 10; format 00/00/0000
CommisionDate	Simple Attribute	Commission Date	Text 10; format 00/00/0000
Length	Simple Attribute	Length of the Ship in Meters	Integer
Width	Simple Attribute	Width of the Ship in Meters	Integer
MastHeight	Simple Attribute	Mast Height of the Ship in Meters	Integer
KeelDepth	Simple Attribute	Keel Depth of the Ship in Meters	Integer
Displacement	Simple Attribute	Displacement of the Ship in Tons	Integer
HomePort	Simple Attribute	Home Port of the Ship	Text 30

Domain Name	Type	Semantic Description	Physical Description
SuperiorIn Command	Simple Attribute	Immediate Superior In Command	Text 50
Overhauls	Group Attribute	Overhauls of the Ship	OverhaulNumber StartDate EndDate ShipyardsName OverhaulDuration
OverhaulNumber	Simple Attribute	Identifying Number Given to each Overhaul	Byte
StartDate	Simple Attribute	Start Date of the Overhaul	Text 10; format 00/00/0000
EndDate	Simple Attribute	End Date of the Overhaul	Text 10; format 00/00/0000
ShipyardsName	Simple Attribute	Shipyards Name Where Overhaul Took Place	Text 50
OverhaulDuration	Simple Attribute	Overhaul Duration in Days	Integer
PlannedManning	Group Attribute	Planned Manning of the Ship	PlannedOfficers PlannedPettyOfficers PlannedEnlisted
PresentManning	Group Attribute	Present Manning of the Ship	PresentOfficers PresentPettyOfficers PresentEnlisted
PlannedOfficers	Simple Attribute	Number of the Planned Officers Onboard Ship	Integer
PlannedPetty Officers	Simple Attribute	Number of the Planned Petty Officers Onboard Ship	Integer

Domain Name	Type	Semantic Description	Physical Description
PlannedEnlisted	Simple Attribute	Number of the Planned Enlisted Onboard Ship	Integer
PresentOfficers	Simple Attribute	Number of the Present Officers Onboard Ship	Integer
PresentPetty Officers	Simple Attribute	Number of the Present Petty Officers Onboard Ship	Integer
PresentEnlisted	Simple Attribute	Number of the Present Enlisted Onboard Ship	Integer

B. DEPARTMENT OBJECT

Domain Name	Type	Semantic Description	Physical Description
DepartmentName	Simple Attribute	Name of a Department Onboard Ship	Text 30
PlannedManning	Group Attribute	Planned Manning of the Department	PlannedOfficers PlannedPettyOfficers PlannedEnlisted
PresentManning	Group Attribute	Present Manning of the Department	PresentOfficers PresentPettyOfficers PresentEnlisted
PlannedOfficers	Simple Attribute	Number of the Planned Officers in Department	Integer
PlannedPetty Officers	Simple Attribute	Number of the Planned Petty Officers in Department	Integer
PlannedEnlisted	Simple Attribute	Number of the Planned Enlisted in Department	Integer
PresentOfficers	Simple Attribute	Number of the Present Officers in Department	Integer
PresentPetty Officers	Simple Attribute	Number of the Present Petty Officers in Department	Integer
PresentEnlisted	Simple Attribute	Number of the Present Enlisted in Department	Integer
PERSONNEL	Semantic Object Attribute	Personnel Assigned to Department	PERSONNEL Object
DIVISION	Semantic Object Attribute	Divisions within the Department	DIVISION Object

C. DIVISION OBJECT

Domain Name	Type	Semantic Description	Physical Description
DivisionName	Simple Attribute	Name of a Division Within a Department	Text 30
PlannedManning	Group Attribute	Planned Manning of the Division	PlannedOfficers PlannedPettyOfficers PlannedEnlisted
PresentManning	Group Attribute	Present Manning of the Division	PresentOfficers PresentPettyOfficers PresentEnlisted
PlannedOfficers	Simple Attribute	Number of the Planned Officers in Division	Integer
PlannedPetty Officers	Simple Attribute	Number of the Planned Petty Officers in Division	Integer
PlannedEnlisted	Simple Attribute	Number of the Planned Enlisted in Division	Integer
PresentOfficers	Simple Attribute	Number of the Present Officers in Division	Integer
PresentPetty Officers	Simple Attribute	Number of the Present Petty Officers in Division	Integer
PresentEnlisted	Simple Attribute	Number of the Present Enlisted in Division	Integer
PERSONNEL	Semantic Object Attribute	Personnel Assigned to Division	PERSONNEL Object
DEPARTMENT	Semantic Object Attribute	Department to which Division Belongs	DEPARTMENT Object

D. PERSONNEL OBJECT

Domain Name	Type	Semantic Description	Physical Description
MilitaryID	Simple Attribute	Military Identification Number of Personnel	Text 10
Name	Group Attribute	First and Last Names of Personnel	FirstName LastName
FirstName	Simple Attribute	First Name of the Personnel	Text 30
LastName	Simple Attribute	Last Name of the Personnel	Text 30
Rank	Simple Attribute	Rank of the Personnel	Text 50
Rating	Simple Attribute	Rating of the Personnel	Text 50; values {Officer, Petty Officer, Enlisted}
DateOfBirth	Simple Attribute	Birth Date of Personnel	Text 10; format 00/00/0000
PlaceOfBirth	Simple Attribute	Place of Birth of Personnel	Text 30
FatherName	Simple Attribute	Father's Name of Personnel	Text 30
MotherName	Simple Attribute	Mother's Name of Personnel	Text 30
ActiveDuty ServiceDate	Simple Attribute	Active Duty Service Date of Personnel	Text 10; format 00/00/0000
DateOfRank	Simple Attribute	Date of Rank of Personnel	Text 10; format 00/00/0000
Gender	Simple Attribute	Gender of Personnel	Text 20; values {Male, Female}

Domain Name	Type	Semantic Description	Physical Description
MaritalStatus	Simple Attribute	Marital Status of Personnel	Text 50; values {Married, Single}
SpouseName	Simple Attribute	Spouse's Name of Personnel	Text 30
NumberOfChildren	Simple Attribute	Number of Children of Personnel	Byte
Address	Group Attribute	Home Address of Personnel	Street City State ZipCode
Street	Simple Attribute	Street Address of Personnel	Text 50
City	Simple Attribute	City where Personnel Lives	Text 30
State	Simple Attribute	State where Personnel Lives	Text 30
ZipCode	Simple Attribute	Zip Code of the Personnel's Address	Text 10; format 00000-9999
PhoneNumber	Simple Attribute	Phone Number of Personnel's House	Text 15; format (000) 000-0000
Specialty	Simple Attribute	Specialty of Personnel	Text 50
Education	Simple Attribute	Education of Personnel	Text 30
Current Assignment	Simple Attribute	Current Assignment of Personnel	Text 100
StartDate	Simple Attribute	Start Date of Current Assignment	Text 10; format 00/00/0000
CabinNumber	Simple Attribute	Cabin Number of Personnel	Text 10

Domain Name	Type	Semantic Description	Physical Description
CabinPhone	Simple Attribute	Cabin Phone of Personnel	Integer; format 000
CoursesToTake	Group Attribute	Military Courses that Personnel should take	TRAINING Object
TRAINING	Semantic Object Attribute	Military Course	TRAINING Object
Previous Assignments	Group Attribute	Previous Assignments of Personnel	Assignment Number Station Position Duration
Assignment Number	Simple Attribute	Identifying Number of Assignment	Byte
Station	Simple Attribute	Station Name of the Previous Assignment	Text 50
Position	Simple Attribute	Position Name of the Previous Assignment	Text 50
Duration	Simple Attribute	Duration of Previous Assignment in Years	Byte
Foreign Languages	Group Attribute	Foreign Languages Known by Personnel	Language Degree
Language	Simple Attribute	Name of the Foreign Language	Text 30
Degree	Simple Attribute	Degree of the Foreign Language	Text 1; values {A, B, C, D, F}
CoursesTaken	Group Attribute	Military Courses that Personnel has taken	TRAINING Object StartDate EndDate Grade

Domain Name	Type	Semantic Description	Physical Description
StartDate	Simple Attribute	Start Date of Course	Text 10; format 00/00/0000
EndDate	Simple Attribute	End Date of Course	Text 10; format 00/00/0000
Grade	Simple Attribute	Course Grade	Byte; values {0 to 100}
DIVISION	Semantic Object Attribute	Division for which Personnel works	DIVISION Object
DEPARTMENT	Semantic Object Attribute	Department for which Personnel works	DEPARTMENT Object

E. TRAINING OBJECT

Domain Name	Type	Semantic Description	Physical Description
CourseName	Simple Attribute	Name of the Military Course	Text 50
TrainingCenter	Simple Attribute	Training Center where Course is given	Text 50
Duration	Simple Attribute	Course Duration in Weeks	Byte
Course Description	Simple Attribute	Brief Description of Course	Text 200
PERSONNEL	Semantic Object Attribute	Personnel who should take the Course	PERSONNEL Object
PERSONNEL	Semantic Object Attribute	Personnel who has taken the Course	PERSONNEL Object

F. EQUIPMENT OBJECT

Domain Name	Type	Semantic Description	Physical Description
SerialNumber	Simple Attribute	Serial Number of Equipment	Text 20
StockNumber	Simple Attribute	Stock Number of Equipment	Text 20
EquipmentName	Simple Attribute	Equipment Name	Text 50
EquipmentType	Simple Attribute	Equipment Type	Text 50
Manufacturer	Simple Attribute	Manufacturer Name	Text 30
Model	Simple Attribute	Equipment Model	Text 30
ProductionDate	Simple Attribute	Production Date of Equipment	Text 10; format 00/00/0000
Location	Simple Attribute	Location of Equipment	Text 10
Runtime	Simple Attribute	Run Time of Equipment in Hours	Long Integer
Failures	Group Attribute	Failures of Equipment	Failure Number Description Diagnosis Failure Date Failure Duration
Failure Number	Simple Attribute	Identifying Number of Failure	Byte
Description	Simple Attribute	Failure Description	Text 100
Diagnosis	Simple Attribute	Diagnosis of Failure	Text 100
FailureDate	Simple Attribute	Failure Date	Text 10; format 00/00/0000
FailureDuration	Simple Attribute	Failure Duration in Hours	Byte

G. OPERATION OBJECT

Domain Name	Type	Semantic Description	Physical Description
ExerciseName	Simple Attribute	Name of Exercise	Text 30
ExerciseType	Simple Attribute	Type of Exercise	Text 50
StartDate	Simple Attribute	Start Date of Exercise	Text 10; format 00/00/0000
EndDate	Simple Attribute	End Date of Exercise	Text 10; format 00/00/0000
Duration	Simple Attribute	Duration of Exercise in Days	Byte
Place	Simple Attribute	Name of the Seas that Exercise took place	Text 100
Events	Group Attribute	Events Executed during Exercise	EventName EventType EventDuration NumberOfEvents
EventName	Simple Attribute	Name of Event	Text 50
EventType	Simple Attribute	Type of Event	Text 30
NumberOfEvents	Simple Attribute	Number of Events	Byte
EventDuration	Simple Attribute	Total Duration of Event in Hours	Integer

Domain Name	Type	Semantic Description	Physical Description
PortVisits	Group Attribute	Ports Visited during Exercise	PortName VisitStartDate VisitEndDate VisitDuration
PortName	Simple Attribute	Name of Port	Text 50
VisitStartDate	Simple Attribute	Start Date of Port Visit	Text 10; format 00/00/0000
VisitEndDate	Simple Attribute	End Date of Port Visit	Text 10; format 00/00/0000
VisitDuration	Simple Attribute	Duration of Port Visit in Days	Byte
Underway Durations	Group Attribute	Underway Duration of Exercise in Hours	DaytimeHours NightTimeHours
DaytimeHours	Simple Attribute	Daytime Underway Hours	Integer
NighttimeHours	Simple Attribute	Nighttime Underway Hours	Integer
CostOfExercise	Group Attribute	Cost of Exercise	FuelCost AmmunitionCost Amortization
FuelCost	Simple Attribute	Cost of Fuel Consumed	Double
AmmunitionCost	Simple Attribute	Cost of Ammunition	Double
Amortization	Simple Attribute	Amortization Cost	Double
CostOfExercise	Simple Attribute	Cost of Exercise	Double

Domain Name	Type	Semantic Description	Physical Description
Helicopter	Group Attribute	Helicopter Stationed Onboard during the Exercise	HeloTailNumber FlyingDuration NumberOfDippings DippingDuration
HeloTailNumber	Simple Attribute	Tail Number of Helicopter	Text 10
FlyingDuration	Simple Attribute	Flying Duration of Helicopter in Hours	Integer
NumberOfDippings	Simple Attribute	Number of Dippings that Helicopter made	Byte
DippingDuration	Simple Attribute	Dipping Duration of Helicopter in Hours	Integer

APPENDIX C: RELATIONAL TABLES

SHIP (InternationalCallSign, ShipName, HullNumber, ShipClass, KeelLayingDate, LaunchDate, CommissionDate, Length, Width, MastHeight, KeelDepth, Displacement, Homeport, SuperiorInCommand, PlannedOfficers, PlannedPettyOfficers, PlannedEnlisted, PresentOfficers, PresentPettyOfficers, PresentEnlisted)

OVERHAULS (InternationalCallSign, OverhaulNumber, StartDate, EndDate, OverhaulDuration, ShipyardName)

PERSONNEL (MilitaryID, FirstName, LastName, DepartmentName, DivisionName, Rank, Rating, DateOfBirth, PlaceOfBirth, FatherName, MotherName, ActiveDutyServiceDate, DateOfRank, Gender, MaritalStatus, SpouseName, NumberOfChildren, Street, City, State, ZipCode, PhoneNumber, Speciality, Education, CurrentAssignment, StartDate, CabinNumber, CabinPhone)

COURSES_TO_TAKE (MilitaryID, CourseName)

COURSES_TAKEN (MilitaryID, CourseName, StartDate, EndDate, Grade)

ASSIGNMENTS (MilitaryID, AssignmentNumber, Station, Position, Duration)

FOREIGN_LANGUAGES (MilitaryID, Language, Degree)

DEPARTMENT (DepartmentName, PlannedOfficers, PlannedPettyOfficers,
PlannedEnlisted, PresentOfficers, PresentPettyOfficers,
PresentEnlisted)

DIVISION (DivisionName, *DepartmentName*, PlannedOfficers, PlannedPettyOfficers,
PlannedEnlisted, PresentOfficers, PresentPettyOfficers, PresentEnlisted)

TRAINING (CourseName, TrainingCenter, CourseDuration, CourseDescription)

OPERATION (ExerciseName, ExerciseType, StartDate, EndDate, Duration, Place,
DaytimeUnderwayHours, NighttimeUnderwayHours, HeloTailNumber,
FlyingDuration, NumberOfDippings, DippingDuration, FuelCost,
AmmunitionCost, Amortization, CostOfExercise)

EVENTS (ExerciseName, EventName, EventType, NumberOfEvents, EventDuration)

PORT_VISITS (ExerciseName, PortName, VisitStartDate, VisitEndDate,
VisitDuration)

EQUIPMENT (SerialNumber, StockNumber, EquipmentName, EquipmentType,
ProductionDate, Manufacturer, Model, Location, Runtime)

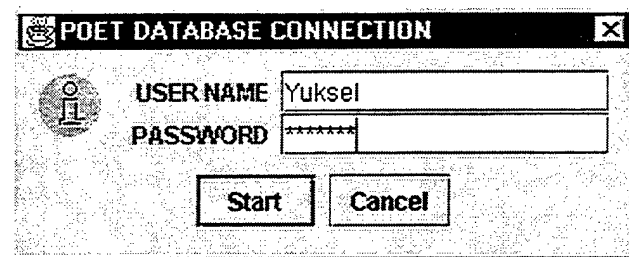
FAILURES (SerialNumber, FailureNumber, FailureDescription, FailureDiagnosis,
FailureDate, FailureDuration)

THIS PAGE INTENTIONALLY LEFT BLANK

THIS PAGE INTENTIONALLY LEFT BLANK

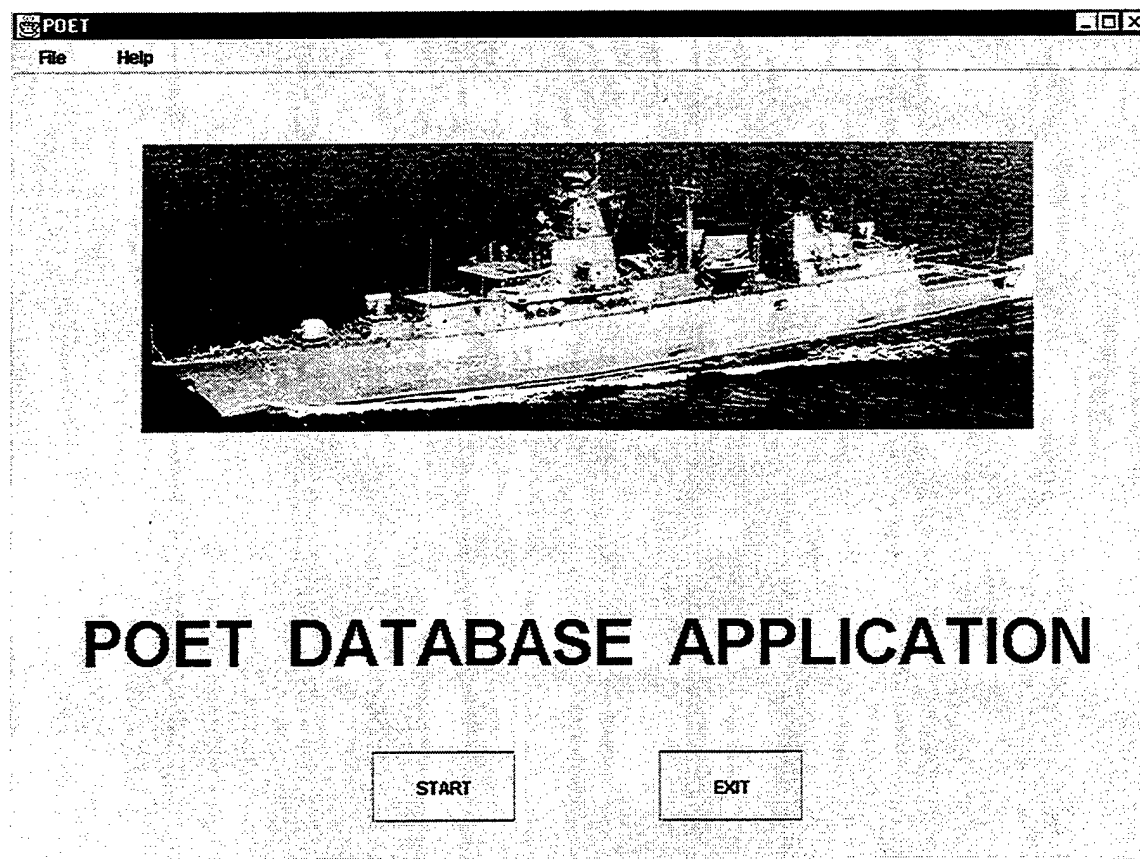
APPENDIX E: APPLICATION PROGRAM SCREEN SHOTS

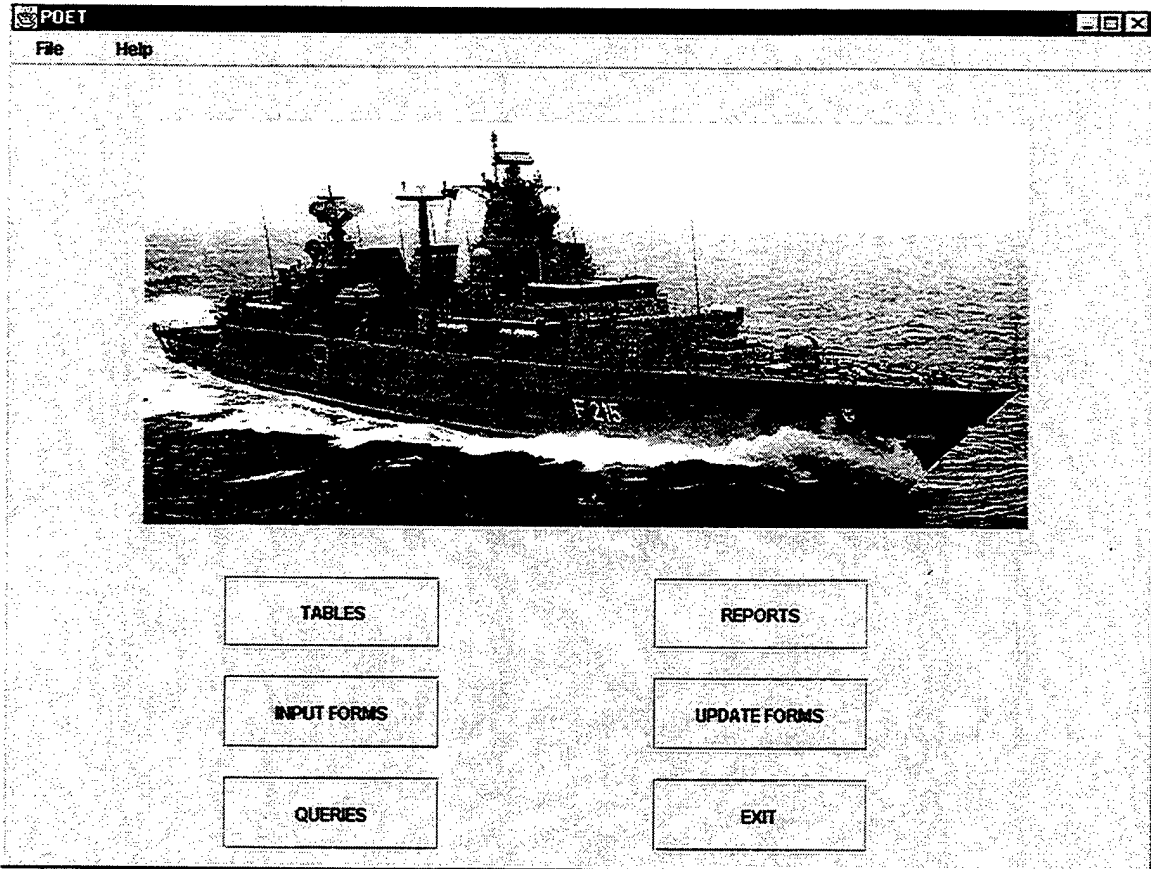
A. CONNECTION PANEL

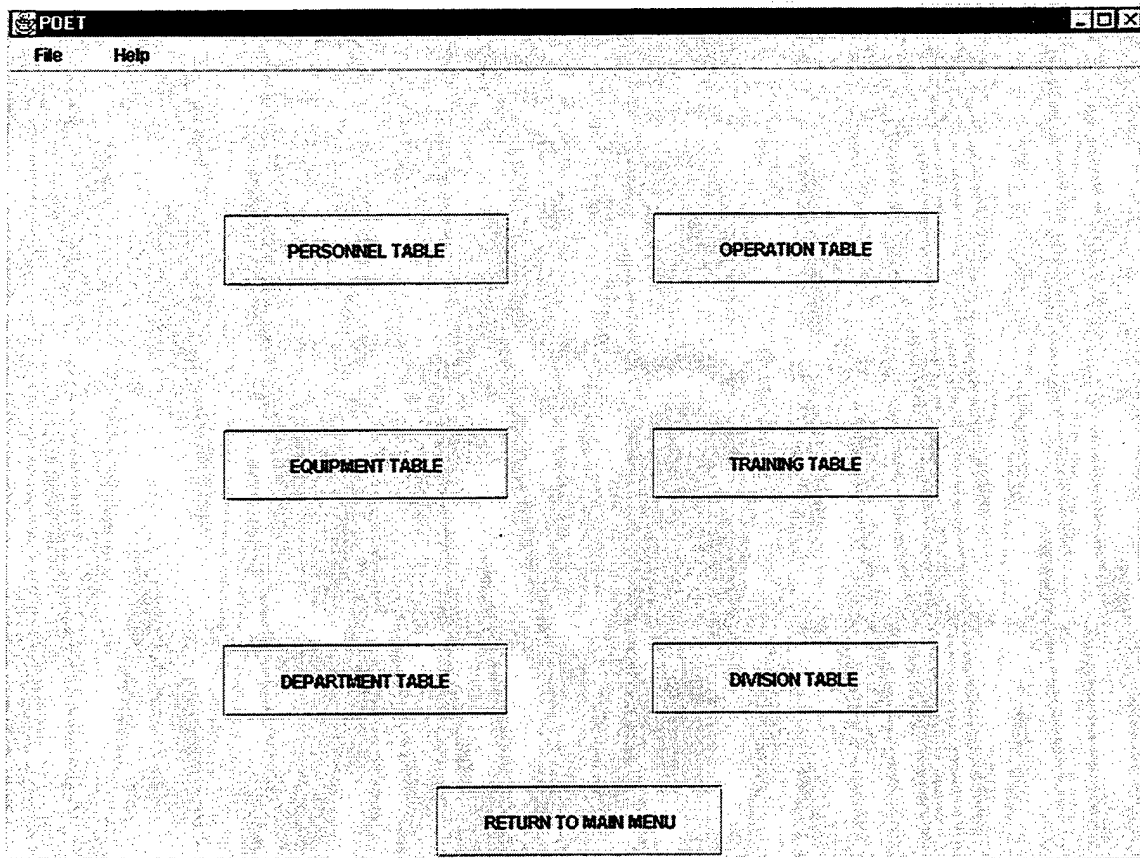


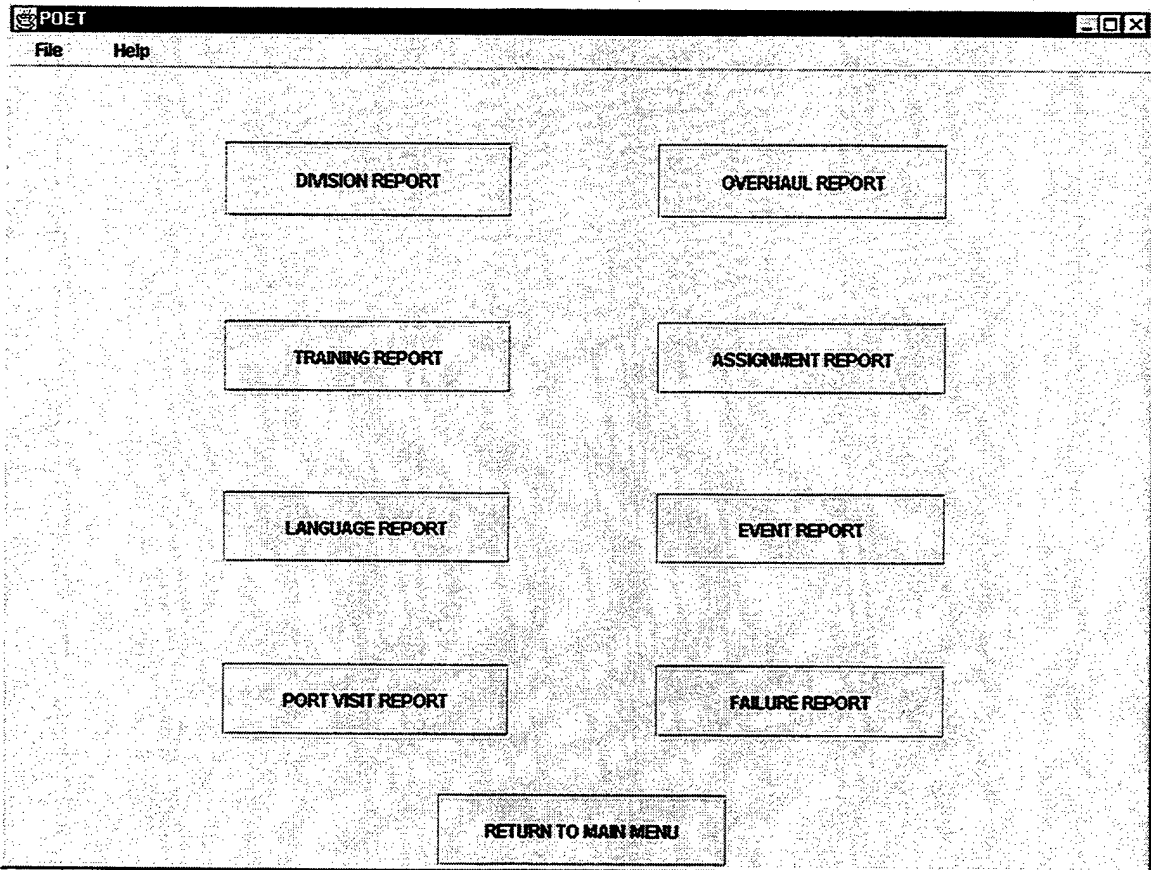
A dialog box titled "POET DATABASE CONNECTION" with a close button (X) in the top right corner. On the left is a circular icon containing a stylized 'i'. To the right of the icon are two input fields: "USER NAME" containing the text "YukseI" and "PASSWORD" containing seven asterisks. Below the input fields are two buttons: "Start" and "Cancel".

B. MENUS / BUTTONS









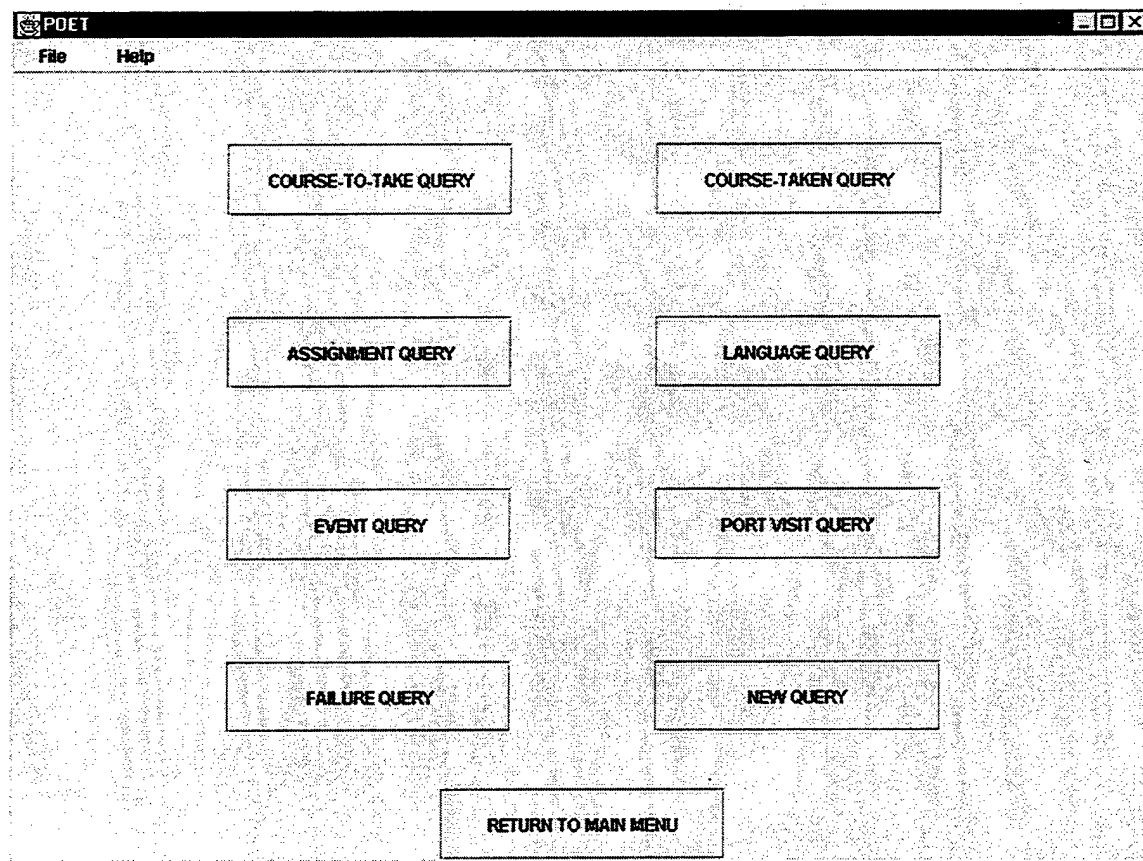
POET File Help

PERSONNEL INPUT FORM	OPERATION INPUT FORM
EQUIPMENT INPUT FORM	TRAINING INPUT FORM
OVERHAUL INPUT FORM	COURSE-TO-TAKE INPUT FO...
COURSE-TAKEN INPUT FORM	ASSIGNMENT INPUT FORM
LANGUAGE INPUT FORM	EVENT INPUT FORM
PORT VISIT INPUT FORM	FAILURE INPUT FORM

RETURN TO MAIN MENU

POET File Help

PERSONNEL UPDATE FORM	OPERATION UPDATE FORM
EQUIPMENT UPDATE FORM	TRAINING UPDATE FORM
OVERHAUL UPDATE FORM	COURSE-TO-TAKE UPDATE ...
COURSE-TAKEN UPDATE FO...	ASSIGNMENT UPDATE FORM
LANGUAGE UPDATE FORM	EVENT UPDATE FORM
PORT VISIT UPDATE FORM	FAILURE UPDATE FORM
RETURN TO MAIN MENU	



C. TABLES

PERSONNEL TABLE						
Inter-Cell Spacing			Row Height			
MilitaryID	FirstName	LastName	DepartmentName	DivisionName	Rank	Rating
19937025	Unal	Aktas	Operations	CIC	LTJG	OFFICER
19947280	Yuksel	Can	Operations	Communications	LTJG	OFFICER
19947358	Ozcan	Altunbulak	Weapons	Anti Air Warfare	LTJG	OFFICER
19915179	Tarkan	Gurul	Weapons	Anti Submarine Warfare	LTJG	OFFICER
19804029	Recep	Gul	Operations	Communications	SENIOR CHIEF PETTY OFFICER	PETTY OFFICER
19804288	Mehmet	Kahraman	Operations	CIC	SENIOR CHIEF PETTY OFFICER	PETTY OFFICER
19742550	Lutfi	Yavuz	Navigation	Administration	COMMANDER	OFFICER
19762966	Mucahit	Sislioglu	Navigation	Administration	COMMANDER	OFFICER
19783055	Selahattin	Deniz	Engineering	Main Propulsion	LTCDR	OFFICER
19793123	Ismet	Hergunsen	Operations	CIC	LTCDR	OFFICER
19823562	Kemal	Evcioğlu	Weapons	Anti Surface Warfare	LIEUTENANT	OFFICER
19833698	Yucel	Atalay	Electronics	Weapons Electronics	LIEUTENANT	OFFICER
19833602	Bulent	Olcay	Navigation	Navigation	LIEUTENANT	OFFICER
19854023	Aydin	Yilmaz	Supply	Supply	LIEUTENANT	OFFICER
19988908	Ahmet	Cankurt	Engineering	Electrical		ENLISTED

OPERATION TABLE					
Inter-Cell Spacing			Row Height		
ExerciseName	ExerciseType	StartDate	EndDate	Duration	Place
DYNAMIC MDX-93	FLEET EXERCISE	10/05/1993	10/22/1993	17	Mediterranean Sea, Aegean Sea
SEA WORM-94	FLEET EXERCISE	06/02/1994	06/30/1994	28	Mediterranean Sea, Aegean Sea, Marmara Sea
FRIENDSHIP-97	SQUADRON EXERCISE	04/12/1997	04/26/1997	14	Black Sea, Marmara Sea
SEA WORM-96	FLEET EXERCISE	06/10/1996	07/09/1996	29	Mediterranean Sea, Aegean Sea, Marmara Sea
DISTANT THUN...	FLEET EXERCISE	04/12/1998	04/28/1998	16	Aegean Sea, Mediterranean Sea
SEA STAR-95	SQUADRON EXERCISE	09/06/1995	09/24/1995	18	Black Sea, Marmara Sea

EQUIPMENT TABLE								
Inter Cell Spacing				Row Height				
SerialNumber	StockNumber	EquipmentName	EquipmentType	ProductionDate	Manufacturer	Model	Location	Runtime
1254698210	100-425-4125	AN/SQS-56	SONAR	10/12/1992	Signal	SQS-56	5C03	17,536
4526947822	1004567106	DECCA	RADAR	02/15/1993	Decca	TN 2001	03D8	19,852
1599630457	2008521138	HF-1	COMMUNICATI...	10/18/1996	Marconi	HF 7003	02H4	5,623
4522106897	2005691255	UHF-3	COMMUNICATI...	12/05/1992	Marconi	SS 12	02H4	14,200
8652496520	3001447845	GPS	NAVIGATION	03/14/1992	Magnavox	MT 900	03D5	18,610

TRAINING TABLE			
CourseName	TrainingCenter	CourseDuration	CourseDescription
Communications	KARAMURSEL TRAINING CENTER	8	Communications Course prepares the ne...
CIC	KARAMURSEL TRAINING CENTER	8	CIC Course gives the necessary backgrou...
Wepaons	KARAMURSEL TRAINING CENTER	6	Weapons Course educates the Wepons Of...
Operations Electronics	DERINCE TRAINING CENTER	32	Operations Electronics Course trains the O...
Weapons Electronics	DERINCE TRAINING CENTER	32	Wepaons Electronics Course trains the Off...
Helicopter Controller	YILDIZLAR TRAINING CENTER	4	Helicopter Controller Course prepares the ...
Commanding Officer	YILDIZLAR TRAINING CENTER	4	Commanding Officer Course educates the ...
Executive Officer	YILDIZLAR TRAINING CENTER	4	Executive Officer Course educates the Exe...
Chief Engineer	DERINCE TRAINING CENTER	6	Chief Engineer Course educates the Chief ...

D. REPORTS

DIVISION REPORT				
DEPARTMENT	DIVISION	OFFICERS	PETTY OFFICERS	ENLISTED
Operations	CIC	2	6	4
	Communications	1	8	0
	Electronic Warfare	1	3	2
Engineering	Main Propulsion	2	8	6
	Electrical	1	6	5
	Damage Control	1	4	5
Weapons	Anti Surface Warfare	1	2	3
	Anti Submarine Warfare	1	3	3
	Anti Air Warfare	1	3	3
	Fire Control	1	2	1
Electronics	Weapons Electronics	2	10	4
	CIC Electronics	1	5	2
	Communications Electronics	1	4	0
Navigation	Administration	2	1	3
	Navigation	1	4	5
	Deck	0	3	6

TRAINING REPORT

FIRST NAME	LAST NAME	COURSE NAME	GRADE
Unal	Aktas	CIC	99
Yuksel	Can	Communications	97
Lutfi	Yavuz	Communications	92
		CIC	90
Mucahit	Sislioglu	Weapons Electronics	97
Selahattin	Deniz	Chief Engineer	94
Ismet	Hergunsen	Helicopter Controller	85
Kemal	Evcioğlu	Weapons	89

ASSIGNMENT REPORT			
FIRST NAME	LAST NAME	STATION	POSITION
Yüksel	Can	TCG YILDIRIM	Communications Officer
Lutfi	Yavuz	TCG SAVASTEPE	Communications Officer
		TCG MUAVENET	CIC Officer
		TUN Headquarters	Operations Officer
Mucahit	Sislioglu	TCG KARAYEL	Fire Control Officer
		TCG FIRTINA	XO
		NATO Headquarters	Intelligence Officer
Selahattin	Deniz	TCG MARTI	Electrical Officer
		TCG KARTAL	Damage Control Officer
		TCH FATIH	Main Propulsion Officer

LANGUAGE REPORT			
FIRST NAME	LAST NAME	LANGUAGE	DEGREE
Unal	Aktas	English	B
Yuksel	Can	English	A
		German	B
Lutfi	Yavuz	English	B
		French	A
Mucahit	Sislioglu	English	A
Selahattin	Deniz	English	C
Kemal	Evcioğlu	German	A
Yucel	Atalay	English	C
		German	B
Bulent	Olcaý	English	A

EXERCISE/EVENT REPORT			
EXERCISE NAME	EVENT NAME	EVENT TYPE	DURATION(Hours)
DYNAMIC MIX-93	SURFEX-310	ANTISURFACE WARFARE	14
	CASEX A-5	ANTISUBMARINE WARFARE	18
	CASEX C-5	ANTISUBMARINE WARFARE	15
SEA WORM-94	SURFEX-310	ANTISURFACE WARFARE	22
	SURFEX-316	ANTISURFACE WARFARE	11
	CASEX C-5	ANTISUBMARINE WARFARE	19
	EWX-320	ANTI-AIR WARFARE	10
	NAVCOMEX-408	COMMUNICATIONS	5
	NAVCOMEX-605	COMMUNICATIONS	4
FRIENDSHIP-97	PASSEX-120	MISCELLANEOUS	12
SEA WORM-96	SURFEX-410	ANTISURFACE WARFARE	21
	CASEX A-5	ANTISUBMARINE WARFARE	15
	NAVCOMEX-405	COMMUNICATIONS	6
	NAVCOMEX-605	COMMUNICATIONS	5
	EWX-231	ELECTRONIC WARFARE	12
DISTANT THUNDER-98	SURFEX-310	ANTISURFACE WARFARE	8

PORT VISIT REPORT			
EXERCISE NAME	PORT NAME	START DATE	END DATE
DYNAMIC MIX-93	NAPOLI	10/10/1993	10/13/1993
SEA WORM-94	IZMIR	06/10/1994	06/13/1994
	ANTALYA	06/18/1994	06/20/1994
	MERSIN	06/23/1994	06/24/1994
FRIENDSHIP-97	VARNA	04/16/1997	04/18/1997
SEA WORM-96	CANAKKALE	06/09/1996	06/11/1996
	MARMARIS	06/15/1996	06/18/1996
	ANTALYA	06/22/1996	06/23/1996
DISTANT THUNDER-98	IZMIR	04/16/1998	04/18/1998
	ISTANBUL	04/22/1998	04/23/1998

EQUIPMENT FAILURE REPORT			
EQUIPMENT NAME	EQUIPMENT TYPE	FAILURE	DURATION(Hours)
AN/SQS-56	SONAR	Stop Transmission	72
		No Signal Reception	95
DECCA	RADAR	No Display	14
		Not Running	19
		Incorrect Target Display	9
HF-1	COMMUNICATIONS	No Transmisson	12
UHF-3	COMMUNICATIONS	No Reception	45
		Not Running	10

OVERHAUL REPORT			
OVERHAUL	START DATE	END DATE	SHIPYARD
1	04/20/1992	07/20/1992	GOLCUK NAVAL SHIPYARD
2	08/12/1994	12/12/1994	GOLCUK NAVAL SHIPYARD
3	05/15/1996	11/15/1996	GOLCUK NAVAL SHIPYARD
4	03/01/1998	05/01/1998	GOLCUK NAVAL SHIPYARD

E. FORMS

PERSONNEL FORM	
Military ID :	19947280
First Name :	Yuksel
Last Name :	Can
Department :	Operations
Division :	Communications
Rank :	LTJG
Rating :	OFFICER
Date Of Birth :	03/05/1972
Place Of Birth :	Kastamonu
Father's Name :	Ismail
Mother's Name :	Ayse
Active Duty Service Date :	08/30/1994
Date Of Rank :	08/30/1997
Gender :	MALE
Marital Status :	MARRIED
Spouse's Name :	Sibel
Number Of Children :	1
Street :	1277 Spruance Road
City :	Monterey
State :	California
Zip Code :	93940-4830
Phone Number :	(831) 372-4408
Speciality :	Computer Science
Education :	MASTER
Current Assignment :	COMMUNICATIONS OFFICER
Start Date :	09/30/1994
Cabin Number :	2K11
Cabin Phone :	244

OPERATION FORM	
Exercise Name :	DISTANT THUNDER-98
Exercise Type :	FLEET EXERCISE
Start Date :	04/12/1998
End Date :	04/28/1998
Duration (Days) :	16
Place (Sea/Ocean):	Aegean Sea, Mediterranean Sea
Daytime Underway Hours :	360
Nighttime Underway Hours :	314
Helo Tail Number :	H45
Helo Flying Time (Hours):	142
Number Of Dippings :	8
Total Dipping Time (Hours):	35
Fuel Cost :	180000.0
Ammunition Cost :	200000.0
Amortization :	220000.0
Cost Of Exercise :	600000.0
<input type="button" value="ADD RECORD"/> <input type="button" value="DELETE RECORD"/> <input type="button" value="UPDATE RECORD"/> <input type="button" value="CANCEL"/>	

EQUIPMENT FORM	
Serial Number :	1254698210
Stock Number :	100-425-4125
Equipment Name :	AN/SQS-56
Equipment Type :	COMMUNICATIONS
Production Date :	10/12/1992
Manufacturer :	Signaal
Equipment Model :	SQS-56
Equipment Location :	5C03
Equipment Runtime (Hours) :	17536

ADD RECORD DELETE RECORD UPDATE RECORD CANCEL

APPENDIX F: APPLICATION PROGRAM CODE

```
//-----  
// File : POETApplication.java  
// Author : LTJG. Yuksel CAN  
// Date : June 22, 1999  
// Description : POETApplication class provides a database frontend  
// application with a graphical user interface to  
// access a Microsoft Access database called POET.mdb  
// that stores and manipulates personnel, operation,  
// equipment, and training information about a ship.  
// The application program uses the JDBC-ODBC Bridge  
// for connecting to the database and swing objects  
// and methods for graphical user interface.  
// Compiler : JDK 1.2.1  
// Comments : POETApplication program can be run as an application  
// or as an applet.  
//-----
```

```
import java.awt.*;  
import java.awt.event.*;  
import java.sql.*;  
import java.util.*;  
import javax.swing.*;  
import javax.swing.event.*;  
import javax.swing.border.*;
```

```
/**  
 * The POETApplication class implements a database frontend  
 * application to access a Microsoft Access database that stores  
 * and manipulates personnel, operation, equipment, and training  
 * information about a ship. The application program uses the  
 * JDBC-ODBC Bridge for connecting to the database and swing objects  
 * and methods for graphical user interface.  
 *  
 * @author LTJG. Yuksel Can  
 */
```

```
public class POETApplication extends JApplet {
```

```
    // Menu Bar  
    JMenuBar menuBar;
```

```
    // Menus  
    JMenu fileMenu;  
    JMenu helpMenu;
```

```
    // CardLayout panels  
    static JPanel deck;  
    static JPanel welcome;
```

```

static JPanel mainMenu;
static JPanel tableMenu;
static JPanel inputFormMenu;
static JPanel updateFormMenu;
static JPanel reportMenu;
static JPanel queryMenu;

//CardLayout Manager
static CardLayout cardManager;

// Menu Items
JMenuItem exit;
JMenuItem contents;
JMenuItem about;

// Main Menu Buttons
JButton tableButton;
JButton inputButton;
JButton updateButton;
JButton reportButton;
JButton queryButton;
JButton stopButton;

// Table Menu Buttons
JButton personnelTable;
JButton operationTable;
JButton equipmentTable;
JButton trainingTable;
JButton departmentTable;
JButton divisionTable;
JButton mainMenuTable;

// Input Form Menu Buttons
JButton personnelInputForm;
JButton operationInputForm;
JButton equipmentInputForm;
JButton trainingInputForm;
JButton overhaulInputForm;
JButton courseToTakeInputForm;
JButton courseTakenInputForm;
JButton assignmentInputForm;
JButton languageInputForm;
JButton eventInputForm;
JButton visitInputForm;
JButton failureInputForm;
JButton mainMenuInputForm;

// Output Form Menu Buttons
JButton personnelUpdateForm;
JButton operationUpdateForm;
JButton equipmentUpdateForm;
JButton trainingUpdateForm;
JButton overhaulUpdateForm;
JButton courseToTakeUpdateForm;

```

```

JButton courseTakenUpdateForm;
JButton assignmentUpdateForm;
JButton languageUpdateForm;
JButton eventUpdateForm;
JButton visitUpdateForm;
JButton failureUpdateForm;
JButton mainMenuUpdateForm;

// Report Menu Buttons
JButton divisionReport;
JButton overhaulReport;
JButton trainingReport;
JButton assignmentReport;
JButton languageReport;
JButton eventReport;
JButton visitReport;
JButton failureReport;
JButton mainMenuReport;

// Query Menu Buttons
JButton courseToTakeQuery;
JButton courseTakenQuery;
JButton assignmentQuery;
JButton languageQuery;
JButton eventQuery;
JButton visitQuery;
JButton failureQuery;
JButton newQuery;
JButton mainMenuQuery;

// Connection Panel object
static ConnectionPanel connectionPanel;

// String array for connectionPanel dialog box
static String[] connectOptionNames = { "Start", "Cancel" };

// Query Window Components
JPanel      queryPanel;
JPanel      leftPanel;
JFrame      queryFrame;
JButton     fetchButton;
JLabel      selectLabel;
JLabel      fromLabel;
JLabel      whereLabel;
JLabel      groupLabel;
JLabel      havingLabel;
JLabel      orderLabel;
JTextArea   selectArea;
JTextArea   fromArea;
JTextArea   whereArea;
JTextArea   groupArea;
JTextArea   havingArea;
JTextArea   orderArea;
JComponent  queryAggregate;

```

```

JScrollPane tableAggregate;

// Static fonts and colors
static Font labelFont;
static Font textFont;
static Font headerFont;

static Color labelColor;
static Color areaColor;
static Color panelColor;
static Color buttonColor;

// Table Model object
JDBCAdapter dataBase;

// JFrame object
static JFrame frame;

/**
 * Method main initializes the frame for the GUI.
 * @param args command line arguments
 * @return void
 */

public static void main( String[] args ) {

    labelFont = new Font("Serif", Font.BOLD, 18);
    textFont = new Font("Serif", Font.BOLD, 16);
    headerFont = new Font("Arial", Font.BOLD, 44);

    labelColor = new Color(170, 200, 170);
    areaColor = new Color(233, 229, 185);
    panelColor = new Color(197, 216, 234);
    buttonColor = new Color(160, 220, 245);

    frame = new JFrame("POET");

    // Handle the window closing event
    frame.addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    });

    // Set Metal Look and Feel
    String metalClassName =
        "javax.swing.plaf.metal.MetalLookAndFeel";

```

```

try {
    UIManager.setLookAndFeel(metalClassName);
    SwingUtilities.updateComponentTreeUI(frame);

    frame.pack();
}

catch (Exception e) {
    JOptionPane.showMessageDialog(null, "ERROR",
        "Metal Look And Feel could not be loaded",
        JOptionPane.ERROR_MESSAGE);
}

POETApplication poetApplication = new POETApplication();

connectionPanel = new ConnectionPanel();

if (JOptionPane.showOptionDialog(null,
    connectionPanel, "POET DATABASE CONNECTION",
    JOptionPane.YES_NO_CANCEL_OPTION,
    JOptionPane.INFORMATION_MESSAGE,
    null, connectOptionNames,
    connectOptionNames[0]) == 0) {

    deck = new JPanel();
    cardManager = new CardLayout();
    deck.setLayout(cardManager);

    frame.getContentPane().add(poetApplication);

    poetApplication.init();

    poetApplication.start();

    // Add panel deck to the applet
    poetApplication.getContentPane().add(deck);

    frame.setSize(800, 600);

    frame.setVisible(true) ;

} // end if

} // end main()

```

```

/**
 * Method init initializes the GUI components and registers
 * event listeners for the menu items.
 * @param args command line arguments
 * @return void
 */

```

```

public void init() {

    // Initialize the menus
    fileMenu = new JMenu("File");
    helpMenu = new JMenu("Help");

    // Initialize the menu items
    exit = new JMenuItem("Exit");
    about = new JMenuItem("About");
    contents = new JMenuItem("Contents");

    // Initialize the welcome panel and its components
    welcome = new JPanel();
    welcome.setLayout(null);
    welcome.setSize(800, 600);
    welcome.setBackground(panelColor);

    String filename = "images/meko.jpg";
    ImageIcon image = new ImageIcon(filename);
    JLabel shipLabel = new JLabel(image);
    shipLabel.setSize(800, 300);
    shipLabel.setLocation(0, 0);

    JLabel header = new JLabel("POET DATABASE APPLICATION",
                               JLabel.CENTER);
    header.setForeground(Color.blue);
    header.setFont(headerFont);
    header.setSize(800, 200);
    header.setLocation(0, 300);

    JButton startButton = new JButton("START");
    startButton.setBackground(panelColor);
    startButton.setForeground(Color.blue);
    startButton.setSize(100, 50);
    startButton.setLocation(250, 475);

    JButton exitButton = new JButton("EXIT");
    exitButton.setBackground(panelColor);
    exitButton.setForeground(Color.blue);
    exitButton.setSize(100, 50);
    exitButton.setLocation(450, 475);

    welcome.add(shipLabel);
    welcome.add(header);
    welcome.add(startButton);
    welcome.add(exitButton);

    deck.add(welcome, "welcome");

    // Initialize the main menu and its components
    mainMenu = new JPanel();
    mainMenu.setLayout(null);
    mainMenu.setSize(800, 600);
    mainMenu.setBackground(panelColor);

```

```

String filename1 = "images/mekol.jpg";
ImageIcon imagel = new ImageIcon(filename1);
JLabel shipLabel1 = new JLabel(imagel);
shipLabel1.setSize(800, 350);
shipLabel1.setLocation(0, 0);

tableButton = new JButton("TABLES");
tableButton.setBackground(panelColor);
tableButton.setForeground(Color.blue);
tableButton.setSize(150, 50);
tableButton.setLocation(150, 350);

inputButton = new JButton("INPUT FORMS");
inputButton.setBackground(panelColor);
inputButton.setForeground(Color.blue);
inputButton.setSize(150, 50);
inputButton.setLocation(150, 420);

updateButton = new JButton("UPDATE FORMS");
updateButton.setBackground(panelColor);
updateButton.setForeground(Color.blue);
updateButton.setSize(150, 50);
updateButton.setLocation(450, 420);

reportButton = new JButton("REPORTS");
reportButton.setBackground(panelColor);
reportButton.setForeground(Color.blue);
reportButton.setSize(150, 50);
reportButton.setLocation(450, 350);

queryButton = new JButton("QUERIES");
queryButton.setBackground(panelColor);
queryButton.setForeground(Color.blue);
queryButton.setSize(150, 50);
queryButton.setLocation(150, 490);

stopButton = new JButton("EXIT");
stopButton.setBackground(panelColor);
stopButton.setForeground(Color.blue);
stopButton.setSize(150, 50);
stopButton.setLocation(450, 490);

mainMenu.add(shipLabel1);
mainMenu.add(tableButton);
mainMenu.add(inputButton);
mainMenu.add(updateButton);
mainMenu.add(reportButton);
mainMenu.add(queryButton);
mainMenu.add(stopButton);

deck.add(mainMenu, "mainMenu");

```

```

// Initialize the table menu and its components
tableMenu = new JPanel();
tableMenu.setLayout(null);
tableMenu.setSize(800, 600);
tableMenu.setBackground(panelColor);

personnelTable = new JButton("PERSONNEL TABLE");
personnelTable.setBackground(buttonColor);
personnelTable.setSize(200, 50);
personnelTable.setLocation(150, 100);

operationTable = new JButton("OPERATION TABLE");
operationTable.setBackground(buttonColor);
operationTable.setSize(200, 50);
operationTable.setLocation(450, 100);

equipmentTable = new JButton("EQUIPMENT TABLE");
equipmentTable.setBackground(buttonColor);
equipmentTable.setSize(200, 50);
equipmentTable.setLocation(150, 250);

trainingTable = new JButton("TRAINING TABLE");
trainingTable.setBackground(buttonColor);
trainingTable.setSize(200, 50);
trainingTable.setLocation(450, 250);

departmentTable = new JButton("DEPARTMENT TABLE");
departmentTable.setBackground(buttonColor);
departmentTable.setSize(200, 50);
departmentTable.setLocation(150, 400);

divisionTable = new JButton("DIVISION TABLE");
divisionTable.setBackground(buttonColor);
divisionTable.setSize(200, 50);
divisionTable.setLocation(450, 400);

mainMenuTable = new JButton("RETURN TO MAIN MENU");
mainMenuTable.setBackground(buttonColor);
mainMenuTable.setSize(200, 50);
mainMenuTable.setLocation(300, 500);

tableMenu.add(personnelTable);
tableMenu.add(operationTable);
tableMenu.add(equipmentTable);
tableMenu.add(trainingTable);
tableMenu.add(departmentTable);
tableMenu.add(divisionTable);
tableMenu.add(mainMenuTable);

deck.add(tableMenu, "tableMenu");

// Initialize the input form menu and its components
inputFormMenu = new JPanel();
inputFormMenu.setLayout(null);

```



```

inputFormMenu.setSize(800, 600);
inputFormMenu.setBackground(panelColor);

personnelInputForm = new JButton("PERSONNEL INPUT FORM");
personnelInputForm.setBackground(buttonColor);
personnelInputForm.setSize(200, 50);
personnelInputForm.setLocation(150, 20);

operationInputForm = new JButton("OPERATION INPUT FORM");
operationInputForm.setBackground(buttonColor);
operationInputForm.setSize(200, 50);
operationInputForm.setLocation(450, 20);

equipmentInputForm = new JButton("EQUIPMENT INPUT FORM");
equipmentInputForm.setBackground(buttonColor);
equipmentInputForm.setSize(200, 50);
equipmentInputForm.setLocation(150, 100);

trainingInputForm = new JButton("TRAINING INPUT FORM");
trainingInputForm.setBackground(buttonColor);
trainingInputForm.setSize(200, 50);
trainingInputForm.setLocation(450, 100);

overhaulInputForm = new JButton("OVERHAUL INPUT FORM");
overhaulInputForm.setBackground(buttonColor);
overhaulInputForm.setSize(200, 50);
overhaulInputForm.setLocation(150, 180);

courseToTakeInputForm = new JButton("COURSE-TO-TAKE INPUT FORM");
courseToTakeInputForm.setBackground(buttonColor);
courseToTakeInputForm.setSize(200, 50);
courseToTakeInputForm.setLocation(450, 180);

courseTakenInputForm = new JButton("COURSE-TAKEN INPUT FORM");
courseTakenInputForm.setBackground(buttonColor);
courseTakenInputForm.setSize(200, 50);
courseTakenInputForm.setLocation(150, 260);

assignmentInputForm = new JButton("ASSIGNMENT INPUT FORM");
assignmentInputForm.setBackground(buttonColor);
assignmentInputForm.setSize(200, 50);
assignmentInputForm.setLocation(450, 260);

languageInputForm = new JButton("LANGUAGE INPUT FORM");
languageInputForm.setBackground(buttonColor);
languageInputForm.setSize(200, 50);
languageInputForm.setLocation(150, 340);

eventInputForm = new JButton("EVENT INPUT FORM");
eventInputForm.setBackground(buttonColor);
eventInputForm.setSize(200, 50);
eventInputForm.setLocation(450, 340);

visitInputForm = new JButton("PORT VISIT INPUT FORM");
visitInputForm.setBackground(buttonColor);

```

```

visitInputForm.setSize(200, 50);
visitInputForm.setLocation(150, 420);

failureInputForm = new JButton("FAILURE INPUT FORM");
failureInputForm.setBackground(buttonColor);
failureInputForm.setSize(200, 50);
failureInputForm.setLocation(450, 420);

mainMenuInputForm = new JButton("RETURN TO MAIN MENU");
mainMenuInputForm.setBackground(buttonColor);
mainMenuInputForm.setSize(200, 50);
mainMenuInputForm.setLocation(300, 500);

inputFormMenu.add(personnelInputForm);
inputFormMenu.add(operationInputForm);
inputFormMenu.add(equipmentInputForm);
inputFormMenu.add(trainingInputForm);
inputFormMenu.add(overhaulInputForm);
inputFormMenu.add(courseToTakeInputForm);
inputFormMenu.add(courseTakenInputForm);
inputFormMenu.add(assignmentInputForm);
inputFormMenu.add(languageInputForm);
inputFormMenu.add(eventInputForm);
inputFormMenu.add(visitInputForm);
inputFormMenu.add(failureInputForm);
inputFormMenu.add(mainMenuInputForm);

deck.add(inputFormMenu, "inputFormMenu");

// Initialize the update form menu and its components
updateFormMenu = new JPanel();
updateFormMenu.setLayout(null);
updateFormMenu.setSize(800, 600);
updateFormMenu.setBackground(panelColor);

personnelUpdateForm = new JButton("PERSONNEL UPDATE FORM");
personnelUpdateForm.setBackground(buttonColor);
personnelUpdateForm.setSize(200, 50);
personnelUpdateForm.setLocation(150, 20);

operationUpdateForm = new JButton("OPERATION UPDATE FORM");
operationUpdateForm.setBackground(buttonColor);
operationUpdateForm.setSize(200, 50);
operationUpdateForm.setLocation(450, 20);

equipmentUpdateForm = new JButton("EQUIPMENT UPDATE FORM");
equipmentUpdateForm.setBackground(buttonColor);
equipmentUpdateForm.setSize(200, 50);
equipmentUpdateForm.setLocation(150, 100);

trainingUpdateForm = new JButton("TRAINING UPDATE FORM");
trainingUpdateForm.setBackground(buttonColor);
trainingUpdateForm.setSize(200, 50);
trainingUpdateForm.setLocation(450, 100);

```

```

overhaulUpdateForm = new JButton("OVERHAUL UPDATE FORM");
overhaulUpdateForm.setBackground(buttonColor);
overhaulUpdateForm.setSize(200, 50);
overhaulUpdateForm.setLocation(150, 180);

courseToTakeUpdateForm = new JButton("COURSE-TO-TAKE UPDATE
                                     FORM");
courseToTakeUpdateForm.setBackground(buttonColor);
courseToTakeUpdateForm.setSize(200, 50);
courseToTakeUpdateForm.setLocation(450, 180);

courseTakenUpdateForm = new JButton("COURSE-TAKEN UPDATE FORM");
courseTakenUpdateForm.setBackground(buttonColor);
courseTakenUpdateForm.setSize(200, 50);
courseTakenUpdateForm.setLocation(150, 260);

assignmentUpdateForm = new JButton("ASSIGNMENT UPDATE FORM");
assignmentUpdateForm.setBackground(buttonColor);
assignmentUpdateForm.setSize(200, 50);
assignmentUpdateForm.setLocation(450, 260);

languageUpdateForm = new JButton("LANGUAGE UPDATE FORM");
languageUpdateForm.setBackground(buttonColor);
languageUpdateForm.setSize(200, 50);
languageUpdateForm.setLocation(150, 340);

eventUpdateForm = new JButton("EVENT UPDATE FORM");
eventUpdateForm.setBackground(buttonColor);
eventUpdateForm.setSize(200, 50);
eventUpdateForm.setLocation(450, 340);

visitUpdateForm = new JButton("PORT VISIT UPDATE FORM");
visitUpdateForm.setBackground(buttonColor);
visitUpdateForm.setSize(200, 50);
visitUpdateForm.setLocation(150, 420);

failureUpdateForm = new JButton("FAILURE UPDATE FORM");
failureUpdateForm.setBackground(buttonColor);
failureUpdateForm.setSize(200, 50);
failureUpdateForm.setLocation(450, 420);

mainMenuUpdateForm = new JButton("RETURN TO MAIN MENU");
mainMenuUpdateForm.setBackground(buttonColor);
mainMenuUpdateForm.setSize(200, 50);
mainMenuUpdateForm.setLocation(300, 500);

updateFormMenu.add(personnelUpdateForm);
updateFormMenu.add(operationUpdateForm);
updateFormMenu.add(equipmentUpdateForm);
updateFormMenu.add(trainingUpdateForm);
updateFormMenu.add(overhaulUpdateForm);
updateFormMenu.add(courseToTakeUpdateForm);
updateFormMenu.add(courseTakenUpdateForm);
updateFormMenu.add(assignmentUpdateForm);

```

```

updateFormMenu.add(languageUpdateForm);
updateFormMenu.add(eventUpdateForm);
updateFormMenu.add(visitUpdateForm);
updateFormMenu.add(failureUpdateForm);
updateFormMenu.add(mainMenuUpdateForm);

deck.add(updateFormMenu, "updateFormMenu");

// Initialize the report menu and its components
reportMenu = new JPanel();
reportMenu.setLayout(null);
reportMenu.setSize(800, 600);
reportMenu.setBackground(panelColor);

divisionReport = new JButton("DIVISION REPORT");
divisionReport.setBackground(buttonColor);
divisionReport.setSize(200, 50);
divisionReport.setLocation(150, 50);

overhaulReport = new JButton("OVERHAUL REPORT");
overhaulReport.setBackground(buttonColor);
overhaulReport.setSize(200, 50);
overhaulReport.setLocation(450, 50);

trainingReport = new JButton("TRAINING REPORT");
trainingReport.setBackground(buttonColor);
trainingReport.setSize(200, 50);
trainingReport.setLocation(150, 170);

assignmentReport = new JButton("ASSIGNMENT REPORT");
assignmentReport.setBackground(buttonColor);
assignmentReport.setSize(200, 50);
assignmentReport.setLocation(450, 170);

languageReport = new JButton("LANGUAGE REPORT");
languageReport.setBackground(buttonColor);
languageReport.setSize(200, 50);
languageReport.setLocation(150, 290);

eventReport = new JButton("EVENT REPORT");
eventReport.setBackground(buttonColor);
eventReport.setSize(200, 50);
eventReport.setLocation(450, 290);

visitReport = new JButton("PORT VISIT REPORT");
visitReport.setBackground(buttonColor);
visitReport.setSize(200, 50);
visitReport.setLocation(150, 410);

failureReport = new JButton("FAILURE REPORT");
failureReport.setBackground(buttonColor);
failureReport.setSize(200, 50);
failureReport.setLocation(450, 410);

```

```

mainMenuReport = new JButton("RETURN TO MAIN MENU");
mainMenuReport.setBackground(buttonColor);
mainMenuReport.setSize(200, 50);
mainMenuReport.setLocation(300, 500);

reportMenu.add(divisionReport);
reportMenu.add(overhaulReport);
reportMenu.add(trainingReport);
reportMenu.add(assignmentReport);
reportMenu.add(languageReport);
reportMenu.add(eventReport);
reportMenu.add(visitReport);
reportMenu.add(failureReport);
reportMenu.add(mainMenuReport);

deck.add(reportMenu, "reportMenu");

// Initialize the query menu and its components
queryMenu = new JPanel();
queryMenu.setLayout(null);
queryMenu.setSize(800, 600);
queryMenu.setBackground(panelColor);

courseToTakeQuery = new JButton("COURSE-TO-TAKE QUERY");
courseToTakeQuery.setBackground(buttonColor);
courseToTakeQuery.setSize(200, 50);
courseToTakeQuery.setLocation(150, 50);

courseTakenQuery = new JButton("COURSE-TAKEN QUERY");
courseTakenQuery.setBackground(buttonColor);
courseTakenQuery.setSize(200, 50);
courseTakenQuery.setLocation(450, 50);

assignmentQuery = new JButton("ASSIGNMENT QUERY");
assignmentQuery.setBackground(buttonColor);
assignmentQuery.setSize(200, 50);
assignmentQuery.setLocation(150, 170);

languageQuery = new JButton("LANGUAGE QUERY");
languageQuery.setBackground(buttonColor);
languageQuery.setSize(200, 50);
languageQuery.setLocation(450, 170);

eventQuery = new JButton("EVENT QUERY");
eventQuery.setBackground(buttonColor);
eventQuery.setSize(200, 50);
eventQuery.setLocation(150, 290);

visitQuery = new JButton("PORT VISIT QUERY");
visitQuery.setBackground(buttonColor);
visitQuery.setSize(200, 50);
visitQuery.setLocation(450, 290);

```

```

failureQuery = new JButton("FAILURE QUERY");
failureQuery.setBackground(buttonColor);
failureQuery.setSize(200, 50);
failureQuery.setLocation(150, 410);

newQuery = new JButton("NEW QUERY");
newQuery.setBackground(buttonColor);
newQuery.setSize(200, 50);
newQuery.setLocation(450, 410);

mainMenuQuery = new JButton("RETURN TO MAIN MENU");
mainMenuQuery.setBackground(buttonColor);
mainMenuQuery.setSize(200, 50);
mainMenuQuery.setLocation(300, 500);

queryMenu.add(courseToTakeQuery);
queryMenu.add(courseTakenQuery);
queryMenu.add(assignmentQuery);
queryMenu.add(languageQuery);
queryMenu.add(eventQuery);
queryMenu.add(visitQuery);
queryMenu.add(failureQuery);
queryMenu.add(newQuery);
queryMenu.add(mainMenuQuery);

deck.add(queryMenu, "queryMenu");

// Add the menu items to the appropriate menus
fileMenu.addSeparator();
fileMenu.add(exit);
fileMenu.addSeparator();

helpMenu.add(contents);
helpMenu.addSeparator();
helpMenu.add(about);

menuBar = new JMenuBar();

// Add menus to the menu bar
menuBar.add(fileMenu);
menuBar.add(helpMenu);

setJMenuBar(menuBar);

// Add action listeners to menu items to handle action events
exit.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});

```

```

contents.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null,
            "FOR HELP, CONSULT WITH LTJG.
            Yuksel Can",
            "HELP TOPICS",
            JOptionPane.INFORMATION_MESSAGE);
    }
});

about.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null,
            "POET DATABASE APPLICATION PROGRAM FOR " +
            "THE TURKISH NAVY FRIGATES, Written By " +
            "LTJG. Yuksel Can", "ABOUT THE PROGRAM",
            JOptionPane.INFORMATION_MESSAGE);
    }
});

// Add action listeners to buttons to handle action events
exitButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});

startButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "mainMenu");
    }
});

tableButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "tableMenu");
    }
});

```

```

inputButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "inputFormMenu");
    }
});

updateButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "updateFormMenu");
    }
});

reportButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "reportMenu");
    }
});

queryButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "queryMenu");
    }
});

stopButton.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
});

personnelTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        personnelTable();
    }
});

```



```

operationTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        operationTable();
    }
});

equipmentTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        equipmentTable();
    }
});

trainingTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        trainingTable();
    }
});

departmentTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        departmentTable();
    }
});

divisionTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        divisionTable();
    }
});

mainMenuTable.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "mainMenu");
    }
});

personnelInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        personnelInputForm();
    }
});

```

```

    }
});

operationInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        operationInputForm();
    }
});

equipmentInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        equipmentInputForm();
    }
});

trainingInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        trainingInputForm();
    }
});

overhaulInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        overhaulInputForm();
    }
});

courseToTakeInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseToTakeInputForm();
    }
});

courseTakenInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseTakenInputForm();
    }
});

assignmentInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {

```

```

        assignmentInputForm();
    }
});

languageInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        languageInputForm();
    }
});

eventInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        eventInputForm();
    }
});

visitInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        visitInputForm();
    }
});

failureInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        failureInputForm();
    }
});

mainMenuInputForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "mainMenu");
    }
});

personnelUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        personnelUpdateForm();
    }
});

```

```

operationUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        operationUpdateForm();
    }
});

equipmentUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        equipmentUpdateForm();
    }
});

trainingUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        trainingUpdateForm();
    }
});

overhaulUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        overhaulUpdateForm();
    }
});

courseToTakeUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseToTakeUpdateForm();
    }
});

courseTakenUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseTakenUpdateForm();
    }
});

assignmentUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        assignmentUpdateForm();
    }
});

```

```

languageUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        languageUpdateForm();
    }
});

eventUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        eventUpdateForm();
    }
});

visitUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        visitUpdateForm();
    }
});

failureUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        failureUpdateForm();
    }
});

mainMenuUpdateForm.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "mainMenu");
    }
});

divisionReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        divisionReport();
    }
});

overhaulReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        overhaulReport();
    }
});

```

```

    }
});

trainingReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        trainingReport();
    }
});

assignmentReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        assignmentReport();
    }
});

languageReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        languageReport();
    }
});

eventReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        eventReport();
    }
});

visitReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        visitReport();
    }
});

failureReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        failureReport();
    }
});

mainMenuReport.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {

```

```

        cardManager.show(deck, "mainMenu");
    }
});

courseToTakeQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseToTakeQuery();
    }
});

courseTakenQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        courseTakenQuery();
    }
});

assignmentQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        assignmentQuery();
    }
});

languageQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        languageQuery();
    }
});

eventQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        eventQuery();
    }
});

visitQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        visitQuery();
    }
});

```

```

failureQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        failureQuery();
    }
});

newQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        showQueryWindow();
    }
});

mainMenuQuery.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        cardManager.show(deck, "mainMenu");
    }
});

} // end init()

/**
 * Method personnelTable retrieves all records from Personnel table.
 * @param none
 * @return void
 */

public void personnelTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Personnel";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

```



```

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(

JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().
    setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new Dimension(spacing,
            spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(
    JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

rowHeightSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int height = ((JSlider) e.getSource()).getValue();
        table.setRowHeight(height);
        table.repaint();
    }
});

JFrame frame = new JFrame("PERSONNEL TABLE");
frame.setSize(800, 600);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());
frame.getContentPane().add(scrollPane, BorderLayout.CENTER );
frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

frame.show();

} // end personnelTable()

```

```

/**
 * Method operationTable retrieves all records from Operation table.
 * @param none
 * @return void
 */

public void operationTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getUrl(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Operation";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

    JPanel controlPanel = new JPanel();

    JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
    controlPanel.add(cellSpacingLabel);

    JSlider cellSpacingSlider = new JSlider(

    JSlider.HORIZONTAL, 0, 10, 1);
    cellSpacingSlider.getAccessibleContext().
        setAccessibleName("Inter-Cell Spacing");
    cellSpacingLabel.setLabelFor(cellSpacingSlider);
    controlPanel.add(cellSpacingSlider);

    cellSpacingSlider.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            int spacing = ((JSlider) e.getSource()).getValue();
            table.setInterCellSpacing(new Dimension(spacing,
            spacing));
            table.repaint();
        }
    });

    JLabel rowHeightLabel = new JLabel("Row Height");
    controlPanel.add(rowHeightLabel);

    JSlider rowHeightSlider = new JSlider(

    JSlider.HORIZONTAL, 5, 100, 20);

```

```

    rowHeightSlider.getAccessibleContext().
        setAccessibleName("Row Height");
    rowHeightLabel.setLabelFor(rowHeightSlider);
    controlPanel.add(rowHeightSlider);

    rowHeightSlider.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            int height = ((JSlider) e.getSource()).getValue();
            table.setRowHeight(height);
            table.repaint();
        }
    });

    JFrame frame = new JFrame("OPERATION TABLE");
    frame.setSize(800, 600);
    frame.setBackground(Color.lightGray);
    frame.getContentPane().setLayout(new BorderLayout());
    frame.getContentPane().add(scrollPane, BorderLayout.CENTER );
    frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

    frame.show();

} // end operationTable()

/**
 * Method equipmentTable retrieves all records from Equipment table.
 * @param none
 * @return void
 */

public void equipmentTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Equipment";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

    JPanel controlPanel = new JPanel();

```

```

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(
    JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().
    setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new Dimension(spacing,
            spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(
    JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

rowHeightSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int height = ((JSlider) e.getSource()).getValue();
        table.setRowHeight(height);
        table.repaint();
    }
});

JFrame frame = new JFrame("EQUIPMENT TABLE");
frame.setSize(800, 600);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());
frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

frame.show();

} // end equipmentTable()

```

```

/**
 * Method trainingTable retrieves all records from Training table.
 * @param none
 * @return void
 */

public void trainingTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Training";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

    JPanel controlPanel = new JPanel();

    JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
    controlPanel.add(cellSpacingLabel);

    JSlider cellSpacingSlider = new JSlider(
        JSlider.HORIZONTAL, 0, 10, 1);
    cellSpacingSlider.getAccessibleContext().
        setAccessibleName("Inter-Cell Spacing");
    cellSpacingLabel.setLabelFor(cellSpacingSlider);
    controlPanel.add(cellSpacingSlider);

    cellSpacingSlider.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            int spacing = ((JSlider) e.getSource()).getValue();
            table.setInterCellSpacing(new Dimension(spacing,
            spacing));
            table.repaint();
        }
    });

    JLabel rowHeightLabel = new JLabel("Row Height");
    controlPanel.add(rowHeightLabel);

    JSlider rowHeightSlider = new JSlider(

```

```

JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

rowHeightSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int height = ((JSlider) e.getSource()).getValue();
        table.setRowHeight(height);
        table.repaint();
    }
});

JFrame frame = new JFrame("TRAINING TABLE");
frame.setSize(800, 600);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());
frame.getContentPane().add(scrollPane, BorderLayout.CENTER );
frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

frame.show();

} // end trainingTable()

```

```

/**
 * Method departmentTable retrieves all records from Department
 * table.
 * @param none
 * @return void
 */

```

```

public void departmentTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Department";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

```

```

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(
    JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().
    setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new Dimension(spacing,
            spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(
    JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

rowHeightSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int height = ((JSlider) e.getSource()).getValue();
        table.setRowHeight(height);
        table.repaint();
    }
});

JFrame frame = new JFrame("DEPARTMENT TABLE");
frame.setSize(800, 600);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());
frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

frame.show();

} // end departmentTable()

```

```

/**
 * Method divisionTable retrieves all records from Division table.
 * @param none
 * @return void
 */

public void divisionTable() {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

    connectionPanel.getDriver(),

    connectionPanel.getUserID(),

    connectionPanel.getPassword());
    dbadapter.connect();

    String tableQuery = "SELECT * FROM Division";
    dbadapter.executeQuery(tableQuery);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

    JPanel controlPanel = new JPanel();

    JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
    controlPanel.add(cellSpacingLabel);

    JSlider cellSpacingSlider = new JSlider(
        JSlider.HORIZONTAL, 0, 10, 1);
    cellSpacingSlider.getAccessibleContext().
        setAccessibleName("Inter-Cell Spacing");
    cellSpacingLabel.setLabelFor(cellSpacingSlider);
    controlPanel.add(cellSpacingSlider);

    cellSpacingSlider.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            int spacing = ((JSlider) e.getSource()).getValue();
            table.setInterCellSpacing(new Dimension(spacing,
            spacing));
            table.repaint();
        }
    });

    JLabel rowHeightLabel = new JLabel("Row Height");
    controlPanel.add(rowHeightLabel);

    JSlider rowHeightSlider = new JSlider(

```



```

JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

rowHeightSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int height = ((JSlider) e.getSource()).getValue();
        table.setRowHeight(height);
        table.repaint();
    }
});

JFrame frame = new JFrame("DIVISION TABLE");
frame.setSize(800, 600);
frame.setBackground(Color.lightGray);
frame.getContentPane().setLayout(new BorderLayout());
frame.getContentPane().add(scrollPane, BorderLayout.CENTER );
frame.getContentPane().add(controlPanel, BorderLayout.NORTH);

frame.show();

} // end divisionTable()

/**
 * Method personnelInputForm adds a new record to the Personnel
 * table.
 * @param none
 * @return void
 */
public void personnelInputForm() {

    final PersonnelForm form = new PersonnelForm("PERSONNEL FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Personnel " +
                "VALUES (" + "" +
                form.militaryIDField.getText() + ", " +
                form.firstNameField.getText() + ", " +
                form.lastNameField.getText() + ", " +

```

```

form.departmentField.getSelectedItemAt() +
", " +
form.divisionField.getSelectedItemAt() +
", " +
form.rankField.getSelectedItemAt() + ", "
+
form.ratingField.getSelectedItemAt() + ",
" +
form.birthDateField.getText() + ", " +
form.birthPlaceField.getText() + ", " +
form.fatherField.getText() + ", " +
form.motherField.getText() + ", " +
form.serviceDateField.getText() + ", "
+
form.rankDateField.getText() + ", " +
form.genderField.getSelectedItemAt() + ",
" +
form.maritalField.getSelectedItemAt() + ",
" +
form.spouseField.getText() + ", " +
form.childrenField.getText() + ", " +
form.streetField.getText() + ", " +
form.cityField.getText() + ", " +
form.stateField.getText() + ", " +
form.zipField.getText() + ", " +
form.phoneField.getText() + ", " +
form.specialityField.getText() + ", " +
form.educationField.getSelectedItemAt() +
", " +
form.assignmentField.getText() + ", " +
form.startDateField.getText() + ", " +
form.cabinNumberField.getText() + ", " +
form.cabinPhoneField.getText() + ")";

updateQuery(insertQuery);

form.dispose();
}
});

form.cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        form.dispose();
    }
});

} // end personnelInputForm()

```

```

/**
 * Method operationInputForm adds a new record to the Operation
 * table.
 * @param none
 * @return void
 */

public void operationInputForm() {

    final OperationForm form = new OperationForm("OPERATION FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Operation " +
                "VALUES (" + "" +
                form.nameField.getText() + ", " +
                form.typeField.getSelectedItem() + ", " +

+
                form.startDateField.getText() + ", " +
                form.endDateField.getText() + ", " +
                form.durationField.getText() + ", " +
                form.placeField.getText() + ", " +
                form.daytimeField.getText() + ", " +
                form.nighttimeField.getText() + ", " +
                form.heloField.getText() + ", " +
                form.flyingField.getText() + ", " +
                form.dippingNumberField.getText() + ", "

+
                form.dippingTimeField.getText() + ", " +
                form.fuelCostField.getText() + ", " +
                form.ammoCostField.getText() + ", " +
                form.amortizationField.getText() + ", " +
                form.costField.getText() + ")";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });
}

```

```

} // end operationInputForm()

/**
 * Method equipmentInputForm adds a new record to the Equipment
 * table.
 * @return void
 */

public void equipmentInputForm() {

    final EquipmentForm form = new EquipmentForm("EQUIPMENT FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Equipment " +
                "VALUES (" + "" +
                form.serialNumberField.getText() + ", " +
+
                form.stockNumberField.getText() + ", " +
+
                form.nameField.getText() + ", " +
                form.typeField.getSelectedItem() + ", " +
+
                form.dateField.getText() + ", " +
                form.manufacturerField.getText() + ", " +
+
                form.modelField.getText() + ", " +
                form.locationField.getText() + ", " +
                form.runtimeField.getText() + ")";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });

} // end equipmentInputForm()

```

```

/**
 * Method trainingInputForm adds a new record to the Training
 * table.
 * @param none
 * @return void
 */

public void trainingInputForm() {

    final TrainingForm form = new TrainingForm("TRAINING FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Training " +
                "VALUES (" + "'" +
                form.nameField.getText() + "', '" +
                form.placeField.getSelectedItem() + "', "
                +
                form.durationField.getText() + ", '" +
                form.descriptionField.getText() + "'"");

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });

} // end trainingInputForm()

```

```

/**
 * Method overhaulInputForm adds a new record to the Overhaul
 * table.
 * @param none
 * @return void
 */

public void overhaulInputForm() {

    final OverhaulForm form = new OverhaulForm("OVERHAUL FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Overhauls " +
                "VALUES (" + "'" +
                form.shipField.getSelectedItem() + "', "
                form.numberField.getText() + ", '" +
                form.startDateField.getText() + "', '" +
                form.endDateField.getText() + "', " +
                form.durationField.getText() + ", '" +
                form.shipyardField.getText() + "')";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });

} // end overhaulInputForm()

```

```

/**
 * Method courseToTakeInputForm adds a new record to the
 * CoursesToTake table.
 * @param none
 * @return void
 */

public void courseToTakeInputForm() {

    final CourseToTakeForm form = new CourseToTakeForm(
        "COURSE-TO-TAKE FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO CoursesToTake " +
                "VALUES (" + "'" +
                form.militaryIDField.getText() + "', '" +
                form.courseField.getText() + "'";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });

} // end courseToTakeInputForm()

```

```

/**
 * Method courseTakenInputForm adds a new record to the
 * CoursesTaken table.
 * @param none
 * @return void
 */

```

```

public void courseTakenInputForm() {

    final CourseTakenForm form = new CourseTakenForm(
        "COURSE-TAKEN FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO CoursesTaken " +
                "VALUES (" + "'" +
                form.militaryIDField.getText() + "', '" +
                form.courseField.getText() + "', '" +
                form.startDateField.getText() + "', '" +
                form.endDateField.getText() + "', " +
                form.gradeField.getText() + ")";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });

} // end courseTakenInputForm()

```

```

/**
 * Method assignmentInputForm adds a new record to the
 * Assignments table.
 * @param none
 * @return void
 */

```

```

public void assignmentInputForm() {

    final AssignmentForm form = new AssignmentForm(
        "ASSIGNMENT FORM");

```



```

form.updateButton.setEnabled(false);

form.deleteButton.setEnabled(false);

form.addButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        String insertQuery = "INSERT INTO Assignments " +
            "VALUES (" + "'" +
            form.militaryIDField.getText() + "', " +
            form.numberField.getText() + ", " +
            form.stationField.getText() + "', " +
            form.positionField.getText() + "', " +
            form.durationField.getText() + ")";

        updateQuery(insertQuery);

        form.dispose();
    }
});

form.cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        form.dispose();
    }
});

} // end assignmentInputForm()

```

```
/**
```

```

* Method languageInputForm adds a new record to the
* ForeignLanguages table.
* @param none
* @return void
*/

```

```

public void languageInputForm() {

    final LanguageForm form = new LanguageForm("LANGUAGE FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {

```

```

+
String insertQuery = "INSERT INTO ForeignLanguages "
"VALUES (" + "'" +
form.militaryIDField.getText() + "', '" +
form.languageField.getSelectedItem() +
"')";
form.degreeField.getSelectedItem() +

```

```

updateQuery(insertQuery);
form.dispose();
}
});

```

```

form.cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        form.dispose();
    }
});

```

```

} // end languageInputForm()

```

```

/**
 * Method eventInputForm adds a new record to the Events table.
 * @param none
 * @return void
 */

```

```

public void eventInputForm() {
    final EventForm form = new EventForm("EXERCISE/EVENT FORM");
    form.updateButton.setEnabled(false);
    form.deleteButton.setEnabled(false);
    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Events " +
                "VALUES (" + "'" +
                form.exerciseField.getText() + "', '" +
                form.eventField.getText() + "', '" +
                form.typeField.getSelectedItem() + "', "
                form.numberField.getText() + ", " +
                form.durationField.getText() + ")";

```

```

        updateQuery(insertQuery);

        form.dispose();
    }
});

form.cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        form.dispose();
    }
});

} // end eventInputForm()

/**
 * Method visitInputForm adds a new record to the PortVisits table.
 * @param none
 * @return void
 */

public void visitInputForm() {

    final PortVisitForm form = new PortVisitForm("PORT VISIT FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO PortVisits " +
                "VALUES (" + "'" +
                form.exerciseField.getText() + "', '" +
                form.portField.getText() + "', '" +
                form.startDateField.getText() + "', '" +
                form.endDateField.getText() + "', " +
                form.durationField.getText() + ")";

            updateQuery(insertQuery);

            form.dispose();
        }
    });
}

```

```

form.cancelButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        form.dispose();
    }
});

} // end visitInputForm()

/**
 * Method failureInputForm adds a new record to the Failures table.
 * @param none
 * @return void
 */

public void failureInputForm() {

    final FailureForm form = new FailureForm(
        "EQUIPMENT FAILURE FORM");

    form.updateButton.setEnabled(false);

    form.deleteButton.setEnabled(false);

    form.addButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            String insertQuery = "INSERT INTO Failures " +
                "VALUES (" + "'" +
                form.serialField.getText() + "', " +
                form.failureField.getText() + ", '" +
                form.descriptionField.getText() + "', '"
                +
                form.diagnosisField.getText() + "', " +
                form.dateField.getText() + ", " +
                form.durationField.getText() + ")";

            updateQuery(insertQuery);

            form.dispose();
        }
    });

    form.cancelButton.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            form.dispose();
        }
    });
}

```

```

    });

} // end failureInputForm()

/**
 * Method personnelUpdateForm retrieves and displays a personnel
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

public void personnelUpdateForm(). {

    JPanel getPanel = new JPanel();
    JLabel getNameLabel = new JLabel("Enter Military ID : ");
    JTextField getNameField = new JTextField(25);

    getPanel.add(getNameLabel);
    getPanel.add(getNameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
                                    "Select Personnel",
                                    JOptionPane.YES_NO_CANCEL_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null, optionNames,
                                    optionNames[0]) == 0) {

        String query = "SELECT * FROM Personnel WHERE " +
                       "MilitaryID = " + "'" +
                       getNameField.getText() + "'";

        ResultSet rs = selectQuery(query);

        try {
            if (rs != null) {

                rs.next();

                final PersonnelForm form = new PersonnelForm(
                    "PERSONNEL FORM");

                form.militaryIDField.setText(rs.getString(1));
                form.firstNameField.setText(rs.getString(2));
                form.lastNameField.setText(rs.getString(3));

                form.departmentField.setSelectedItem(rs.getString(4));

                form.divisionField.setSelectedItem(rs.getString(5));

                form.rankField.setSelectedItem(rs.getString(6));
            }
        }
    }
}

```

```

form.ratingField.setSelectedItem(rs.getString(7));
    form.birthDateField.setText(rs.getString(8));
    form.birthPlaceField.setText(rs.getString(9));
    form.fatherField.setText(rs.getString(10));
    form.motherField.setText(rs.getString(11));

form.serviceDateField.setText(rs.getString(12));
    form.rankDateField.setText(rs.getString(13));

form.genderField.setSelectedItem(rs.getString(14));

form.maritalField.setSelectedItem(rs.getString(15));
    form.spouseField.setText(rs.getString(16));
    form.childrenField.setText(rs.getString(17));
    form.streetField.setText(rs.getString(18));
    form.cityField.setText(rs.getString(19));
    form.stateField.setText(rs.getString(20));
    form.zipField.setText(rs.getString(21));
    form.phoneField.setText(rs.getString(22));
    form.specialityField.setText(rs.getString(23));

form.educationField.setSelectedItem(rs.getString(24));
    form.assignmentField.setText(rs.getString(25));
    form.startDateField.setText(rs.getString(26));

form.cabinNumberField.setText(rs.getString(27));
    form.cabinPhoneField.setText(rs.getString(28));

    form.militaryIDField.setEditable(false);

    form.addButton.setEnabled(false);

    form.updateButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent
        e)
        {
            String updateString = "UPDATE

Personnel SET " +

                                "FirstName = " + "'" +
form.firstNameField.getText() + "', " +

                                "LastName = " + "'" +
form.lastNameField.getText()
+ "', " +

                                "DepartmentName = " + "'" +
form.departmentField.getSelectedItemAt() + "', " +

                                "DivisionName = " + "'" +
form.divisionField.getSelectedItemAt() + "', " +

                                "Rank = " + "'" +

```

```

form.rankField.getSelectedItemAt() + "'", " +
form.ratingField.getSelectedItemAt() + "'", " +
form.birthDateField.getText() + "'", " +
form.birthPlaceField.getText() + "'", " +
"', " +
"', " +
"' " +
form.serviceDateField.getText() + "'", " +
+ "'", " +
form.genderField.getSelectedItemAt() + "'", " +
form.maritalField.getSelectedItemAt() + "'", " +
"', " +
+ " " +
"', " +
"', " +
"', " +
"', " +
"', " +
"', " +
form.specialityField.getText() + "'", " +
"Rating = " + "'" +
"DateOfBirth = " + "'" +
"PlaceOfBirth = " + "'" +
"FatherName = " + "'" +
form.fatherField.getText() +
"MotherName = " + "'" +
form.motherField.getText() +
"ActiveDutyServiceDate = " +
"DateOfRank = " + "'" +
form.rankDateField.getText()
"Gender = " + "'" +
"MaritalStatus = " + "'" +
"SpouseName = " + "'" +
form.spouseField.getText() +
"NumberOfChildren = " +
form.childrenField.getText()
"Street = " + "'" +
form.streetField.getText() +
"City = " + "'" +
form.cityField.getText() +
"State = " + "'" +
form.stateField.getText() +
"ZipCode = " + "'" +
form.zipField.getText() +
"PhoneNumber = " + "'" +
form.phoneField.getText() +
"Speciality = " + "'" +
"Education = " + "'" +

```

```

form.educationField.getSelectedItem() + "'", " +
+
form.assignmentField.getText() + "'", " +
form.startDateField.getText() + "'", " +
form.cabinNumberField.getText() + "'", " +
form.cabinPhoneField.getText() +
+
form.militaryIDField.getText() + "'";

```

```

updateQuery(updateString);

```

```

form.dispose();
}
});

```

```

ActionListener()
{
    public void actionPerformed(ActionEvent
e)
{
    String deleteString = "DELETE FROM
Personnel " +
= " + "'" +
form.militaryIDField.getText() + "'";

```

```

updateQuery(deleteString);

```

```

form.dispose();
}
});

```

```

form.cancelButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
{
    form.dispose();
}
});

```



```

        });
    }
    else {
        JOptionPane.showMessageDialog(this,
            "Unable to find record in
            database",
            "Record Not Found",
            JOptionPane.ERROR_MESSAGE);

        } // end if

    } // end try

    catch (SQLException e) {
        JOptionPane.showMessageDialog(this,
            "SQL Exception : " +
            e.getMessage(),
            "SQL ERROR",
            JOptionPane.ERROR_MESSAGE);

        } // end catch

    } // end if

} // end personnelUpdateForm()

/**
 * Method operationUpdateForm retrieves and displays an exercise
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

public void operationUpdateForm() {

    JPanel getPanel = new JPanel();
    JLabel getNameLabel = new JLabel("Enter Exercise Name : ");
    JTextField getNameField = new JTextField(25);

    getPanel.add(getNameLabel);
    getPanel.add(getNameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Exercise",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,

```

```

        optionNames[0]) == 0) {
String query = "SELECT * FROM Operation WHERE " +
               "ExerciseName = " + "'" +
               getNameField.getText() + "'";

ResultSet rs = selectQuery(query);

try {
    if (rs != null) {

        rs.next();

        final OperationForm form = new OperationForm(
            "OPERATION FORM");

        form.nameField.setText(rs.getString(1));

        form.typeField.setSelectedItem(rs.getString(2))
        form.startDateField.setText(rs.getString(3));
        form.endDateField.setText(rs.getString(4));
        form.durationField.setText(rs.getString(5));
        form.placeField.setText(rs.getString(6));
        form.daytimeField.setText(rs.getString(7));
        form.nighttimeField.setText(rs.getString(8));
        form.heloField.setText(rs.getString(9));
        form.flyingField.setText(rs.getString(10));
        form.dippingNumberField.setText(rs.getString(11
        ));

        form.dippingTimeField.setText(rs.getString(12))
        form.fuelCostField.setText(rs.getString(13));
        form.ammoCostField.setText(rs.getString(14));

        form.amortizationField.setText(rs.getString(15)
        );

        form.costField.setText(rs.getString(16));

        form.nameField.setEditable(false);

        form.addButton.setEnabled(false);

        form.updateButton.addActionListener(new
        ActionListener()
        {
            public void actionPerformed(ActionEvent
            e)
            {
                String updateString = "UPDATE
                "ExerciseType = " + "'" +
                form.typeField.getSelectedItem() + "', " +
                "StartDate = " + "'" +

```

```

form.startDateField.getText() + "'", " +
+ "'", " +
+ ", " +
"', " +
+ ", " +
+
form.nighttimeField.getText() + ", " +
"', " +
", " +
form.dippingNumberField.getText() + ", " +
form.dippingTimeField.getText() + ", " +
+ ", " +
+ ", " +
form.amortizationField.getText() + ", " +
"' " +
" ";

"EndDate = " + "'" +
form.endDateField.getText()

"Duration = " +
form.durationField.getText()

"Place = " + "'" +
form.placeField.getText() +
"DaytimeUnderwayHours = " +
form.daytimeField.getText()

"NighttimeUnderwayHours = "

"HeloTailNumber = " + "'" +
form.heloField.getText() +
"FlyingDuration = " +
form.flyingField.getText() +
"NumberOfDippings = " +
"DippingDuration = " +
"FuelCost = " +
form.fuelCostField.getText()
"AmmunitionCost = " +
form.ammoCostField.getText()
"Amortization = " +
"CostOfExercise = " +
form.costField.getText() +
" WHERE ExerciseName = " +
form.nameField.getText() +

updateQuery(updateString);
form.dispose();
}
});

```

```

ActionListener()
    form.deleteButton.addActionListener(new
    {
        public void actionPerformed(ActionEvent
e)
        {
            String deleteString = "DELETE FROM
            "WHERE ExerciseName =
            form.nameField.getText() + "'";

            updateQuery(deleteString);
            form.dispose();
        }
    });

ActionListener()
    form.cancelButton.addActionListener(new
    {
        public void actionPerformed(ActionEvent
e)
        {
            form.dispose();
        }
    });

}
else {
    JOptionPane.showMessageDialog(this,
    database",
    "Unable to find record in
    "Record Not Found",
    JOptionPane.ERROR_MESSAGE);

} // end if

} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
    "SQL Exception : " +
    e.getMessage(),
    "SQL ERROR",
    JOptionPane.ERROR_MESSAGE);

} // end catch

} // end if

} // end operationUpdateForm()

```

```

/**
 * Method equipmentUpdateForm retrieves and displays an equipment
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

public void equipmentUpdateForm() {

    JPanel getPanel = new JPanel();
    JLabel getNameLabel = new JLabel("Enter the Serial Number : ");
    JTextField getNameField = new JTextField(25);

    getPanel.add(getNameLabel);
    getPanel.add(getNameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
                                    "Select Equipment",
                                    JOptionPane.YES_NO_CANCEL_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null, optionNames,
                                    optionNames[0]) == 0) {

        String query = "SELECT * FROM Equipment WHERE " +
                       "SerialNumber = " + "'" +
                       getNameField.getText() + "'";

        ResultSet rs = selectQuery(query);

        try {
            if (rs != null) {

                rs.next();

                final EquipmentForm form = new EquipmentForm(
                    "EQUIPMENT FORM");

                form.serialNumberField.setText(rs.getString(1))
                ;
                form.stockNumberField.setText(rs.getString(2));
                form.nameField.setText(rs.getString(3));

                form.typeField.setSelectedItem(rs.getString(4))
                ;
                form.dateField.setText(rs.getString(5));

                form.manufacturerField.setText(rs.getString(6))
                ;
                form.modelField.setText(rs.getString(7));
                form.locationField.setText(rs.getString(8));
                form.runtimeField.setText(rs.getString(9));
            }
        }
    }
}

```

```

form.serialNumberField.setEditable(false);

form.addButton.setEnabled(false);

form.updateButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String updateString = "UPDATE
Equipment SET " +
        "StockNumber = " + "'" +
form.stockNumberField.getText() + "', " +
        "EquipmentName = " + "'" +
form.nameField.getText() +
        "EquipmentType = " + "'" +
form.typeField.getSelectedItem() + "', " +
        "ProductionDate = " + "'" +
form.dateField.getText() +
        "Manufacturer = " + "'" +
form.manufacturerField.getText() + "', " +
        "Model = " + "'" +
form.modelField.getText() +
        "Location = " + "'" +
form.locationField.getText()
+ "', " +
        "Runtime = " +
form.runtimeField.getText()
+
        " WHERE SerialNumber = " +
form.serialNumberField.getText() + "'";

        updateQuery(updateString);
        form.dispose();
    }
});

form.deleteButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)

```

```

        {
            String deleteString = "DELETE FROM
Equipment " +
                                "WHERE
SerialNumber = " + "'" +
                                form.serialNumberField.getText() + "'";
                                updateQuery(deleteString);
                                form.dispose();
        }
    });

    form.cancelButton.addActionListener(new
ActionListener()
    {
        public void actionPerformed(ActionEvent
e)
        {
            form.dispose();
        }
    });

}

else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);

} // end if

} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);

} // end catch

} // end if

} // end equipmentUpdateForm()

```

```

/**
 * Method trainingUpdateForm retrieves and displays a course
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

public void trainingUpdateForm() {

    JPanel getPanel = new JPanel();
    JLabel getNameLabel = new JLabel("Enter the Course Name : ");
    JTextField getNameField = new JTextField(25);

    getPanel.add(getNameLabel);
    getPanel.add(getNameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
                                    "Select Course",
                                    JOptionPane.YES_NO_CANCEL_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null, optionNames,
                                    optionNames[0]) == 0) {

        String query = "SELECT * FROM Training WHERE " +
                       "CourseName = " + "'" +
                       getNameField.getText() + "'";

        ResultSet rs = selectQuery(query);

        try {
            if (rs != null) {

                rs.next();

                final TrainingForm form = new TrainingForm(
                    "TRAINING FORM");

                form.nameField.setText(rs.getString(1));

                form.placeField.setSelectedItem(rs.getString(2)
                );

                form.durationField.setText(rs.getString(3));
                form.descriptionField.setText(rs.getString(4));

                form.nameField.setEditable(false);

                form.addButton.setEnabled(false);

                form.updateButton.addActionListener(new
                ActionListener()
                {
                    public void actionPerformed(ActionEvent
                    e)

```



```

        {
            String updateString = "UPDATE
Training SET " +
                "TrainingCenter = " + "\"" +
form.placeField.getSelectedItem() +
                "\", " + "CourseDuration = "
+
                form.durationField.getText()
+ " ", " +
                "CourseDescription = " + "\""
+
form.descriptionField.getText() + "\"" +
                " WHERE CourseName = " + "\""
+
                form.nameField.getText() +
                "\"";

```

```

                updateQuery(updateString);

```

```

                form.dispose();
            }
        });

```

```

form.deleteButton.addActionListener(new
ActionListener()

```

```

{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM

```

```

Training " +

```

```

= " + "\"" +

```

```

        form.nameField.getText() + "\"";

```

```

                "WHERE CourseName

```

```

                updateQuery(deleteString);

```

```

                form.dispose();
            }
        });

```

```

form.cancelButton.addActionListener(new
ActionListener()

```

```

{
    public void actionPerformed(ActionEvent
e)
    {
        form.dispose();
    }
}

```

```

        });
    }
    else {
        JOptionPane.showMessageDialog(this,
            "Unable to find record in
            database",
            "Record Not Found",
            JOptionPane.ERROR_MESSAGE);
    } // end if
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
} // end catch
} // end if
} // end trainingUpdateForm()

```

```
/**
```

```

 * Method overhaulUpdateForm retrieves and displays an overhaul
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

```

```

public void overhaulUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel signLabel = new JLabel("Enter the Ship's Call Sign : ");
    JTextField signField = new JTextField(25);

    JLabel numberLabel = new JLabel("Enter the Overhaul Number : ");
    JTextField numberField = new JTextField(25);

    labelPanel.setLayout(new GridLayout(0, 1));
    labelPanel.add(signLabel);
    labelPanel.add(numberLabel);
}

```

```

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(signField);
fieldPanel.add(numberField);

getPanel.setLayout(new BorderLayout(getPanel, BorderLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Overhaul",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM Overhauls WHERE " +
                  "InternationalCallSign = " +
    "" +
    " +
                  signField.getText() + " AND
                  OverhaulNumber = " +
                  numberField.getText();

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final OverhaulForm form = new OverhaulForm(
                "OVERHAUL FORM");

            form.shipField.setSelectedItem(rs.getString(1))
            ;
            form.numberField.setText(rs.getString(2));
            form.startDateField.setText(rs.getString(3));
            form.endDateField.setText(rs.getString(4));
            form.durationField.setText(rs.getString(5));
            form.shipyardField.setText(rs.getString(6));

            form.shipField.setEditable(false);

            form.numberField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
            ActionListener()
            {

```

```

        public void actionPerformed(ActionEvent
        e)
        {
            String updateString = "UPDATE
Overhaul SET " +
            "StartDate = " + "'" +
form.startDateField.getText() + "', " +
            "EndDate = " + "'" +
+ "'", " +
form.endDateField.getText()
            "OverhaulDuration = " +
+ ", " +
form.durationField.getText()
            "ShipyardName = " + "'" +
+ "'" +
form.shipyardField.getText()

            " WHERE
InternationalCallSign = " + "'" +
form.shipField.getSelectedItem() + "' AND " +
            "OverhaulNumber = " +
form.numberField.getText();

            updateQuery(updateString);
            form.dispose();
        }
    });

    form.deleteButton.addActionListener(new
ActionListener()
    {
        public void actionPerformed(ActionEvent
        e)
        {
            String deleteString = "DELETE FROM
Overhauls " +
            "WHERE
InternationalCallSign = " + "'" +
form.shipField.getSelectedItem() +
            "' AND OverhaulNumber
= " +
form.numberField.getText();

            updateQuery(deleteString);
            form.dispose();
        }
    });

```

```

        form.cancelButton.addActionListener(new
ActionListener()
    {
        public void actionPerformed(ActionEvent
e)
    {
        form.dispose();
    }
    });
}
else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);
} // end if
} // end try
catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
} // end catch
} // end if
} // end overhaulUpdateForm()

```

```

/**
 * Method courseToTakeUpdateForm retrieves and displays a course
 * record that could be taken by a person, which can then be
 * modified or deleted.
 * @param none
 * @return void
 */

```

```

public void courseToTakeUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel idLabel = new JLabel("Enter the Military ID : ");

```

```

final JTextField idField = new JTextField(25);

JLabel nameLabel = new JLabel("Enter the Course Name : ");
final JTextField nameField = new JTextField(25);

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(idLabel);
labelPanel.add(nameLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(idField);
fieldPanel.add(nameField);

getPanel.setLayout(new BoxLayout(getPanel, BoxLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select CourseToTake",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM CoursesToTake WHERE " +
                  "MilitaryID = " + "'" +
                  idField.getText() + "' AND "

                  "CourseName = " + "'" +
                  nameField.getText() + "'";

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final CourseToTakeForm form = new
            CourseToTakeForm("COURSE-TO-TAKE FORM");

            form.militaryIDField.setText(rs.getString(1));
            form.courseField.setText(rs.getString(2));

            form.militaryIDField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent
                e)

```

```

CoursesToTake SET " +
"" +
+
+
{
    String updateString = "UPDATE
        "CourseName = " + "" +
        form.courseField.getText() +
        " WHERE MilitaryID = " + ""
        idField.getText() + " AND "
        "CourseName = " + "" +
        nameField.getText() + "";
    updateQuery(updateString);
    form.dispose();
}
});

form.deleteButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM " +
            "CoursesToTake " +
            "WHERE MilitaryID = "
            idField.getText() + "
            "CourseName = " + ""
            nameField.getText() +

        updateQuery(deleteString);
        form.dispose();
    }
});

form.cancelButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        form.dispose();
    }
});

```

```

    }

    else {
        JOptionPane.showMessageDialog(this,
            "Unable to find record in
            database",
            "Record Not Found",
            JOptionPane.ERROR_MESSAGE);

        } // end if

    } // end try

    catch (SQLException e) {
        JOptionPane.showMessageDialog(this,
            "SQL Exception : " +
            e.getMessage(),
            "SQL ERROR",
            JOptionPane.ERROR_MESSAGE);

        } // end catch

    } // end if

} // end courseToTakeUpdateForm()

```

```
/**
```

```

* Method courseTakenUpdateForm retrieves and displays a course
* record that was taken by a person, which can then be
* modified or deleted.
* @param none
* @return void
*/

```

```

public void courseTakenUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel idLabel = new JLabel("Enter the Military ID : ");
    final JTextField idField = new JTextField(25);

    JLabel nameLabel = new JLabel("Enter the Course Name : ");
    final JTextField nameField = new JTextField(25);

    labelPanel.setLayout(new GridLayout(0, 1));
    labelPanel.add(idLabel);
    labelPanel.add(nameLabel);

```



```

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(idField);
fieldPanel.add(nameField);

getPanel.setLayout(new BorderLayout(getPanel, BorderLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select CourseTaken",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM CoursesTaken WHERE " +
                  "MilitaryID = " + "'" +
                  idField.getText() + "' AND " +
                  "CourseName = " + "'" +
                  nameField.getText() + "'";

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final CourseTakenForm form = new
                CourseTakenForm("COURSE-TAKEN FORM");

            form.militaryIDField.setText(rs.getString(1));
            form.courseField.setText(rs.getString(2));
            form.startDateField.setText(rs.getString(3));
            form.endDateField.setText(rs.getString(4));
            form.gradeField.setText(rs.getString(5));

            form.militaryIDField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
                ActionListener()
            {
                public void actionPerformed(ActionEvent
                    e)
                {
                    String updateString = "UPDATE
                        CoursesTaken SET " +
                        "CourseName = " + "'" +

```

```

form.courseField.getText() + "'", "
+
    "StartDate = " + "'" +
form.startDateField.getText() + "'", " +
+ "'", " +
+
+
    "EndDate = " + "'" +
form.endDateField.getText()
    "Grade = " +
form.gradeField.getText() +
    " WHERE MilitaryID = " + "'"
idField.getText() + "' AND "
    "CourseName = " + "'" +
nameField.getText() + "'";
updateQuery(updateString);
form.dispose();
}
});

```

```

ActionListener()
form.deleteButton.addActionListener(new
{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM " +
            "CoursesTaken " +
            "WHERE MilitaryID = "
+ "'" +
AND " +
+
            idField.getText() + "'
            "CourseName = " + "'"
            nameField.getText() +

        updateQuery(deleteString);
        form.dispose();
    }
});

```

```

ActionListener()
form.cancelButton.addActionListener(new
{
    public void actionPerformed(ActionEvent
e)

```

```

        {
            form.dispose();
        }
    });

}

else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);

    } // end if

} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);

    } // end catch

} // end if

} // end courseTakenUpdateForm()

```

```
/**
```

```

* Method assignmentUpdateForm retrieves and displays a previous
* assignment record for a person, which can then be modified
* or deleted.
* @param none
* @return void
*/

```

```
public void assignmentUpdateForm() {
```

```

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

```

```

    JLabel idLabel = new JLabel("Enter the Military ID : ");
    final JTextField idField = new JTextField(25);

```

```

    JLabel numberLabel = new JLabel("Enter the Assignment Number :

```

```
");
```

```

    final JTextField numberField = new JTextField(25);

```

```

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(idLabel);
labelPanel.add(numberLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(idField);
fieldPanel.add(numberField);

getPanel.setLayout(new BoxLayout(getPanel, BoxLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Previous Assignment",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM Assignments WHERE " +
                  "MilitaryID = " + "'" +
                  idField.getText() + "' AND "

                  "AssignmentNumber = " +
                  numberField.getText();

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final AssignmentForm form = new AssignmentForm(
                "ASSIGNMENT FORM");

            form.militaryIDField.setText(rs.getString(1));
            form.numberField.setText(rs.getString(2));
            form.stationField.setText(rs.getString(3));
            form.positionField.setText(rs.getString(4));
            form.durationField.setText(rs.getString(5));

            form.militaryIDField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent
                e)
                {

```

```

Assignments SET " +
    "AssignmentNumber = " +
    form.numberField.getText() +
    ", " +
    "Station = " + "'" +
    form.stationField.getText() +
    "' " +
    "Position = " + "'" +
    form.positionField.getText() +
    "' " +
    "Duration = " +
    form.durationField.getText() +
    " " +
    " WHERE MilitaryID = " + "'" +
    idField.getText() + "' AND " +
    "AssignmentNumber = " +
    numberField.getText();

updateQuery(updateString);

form.dispose();
});

```

```

form.deleteButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM " +
            "Assignments " +
            "WHERE MilitaryID = " +
            idField.getText() + "' AND " +
            "AssignmentNumber = " +
            numberField.getText();

        updateQuery(deleteString);

        form.dispose();
    }
});

```

```

form.cancelButton.addActionListener(new
ActionListener()

```

```

        {
            public void actionPerformed(ActionEvent
            e)
            {
                form.dispose();
            }
        });
    }

    else {
        JOptionPane.showMessageDialog(this,
            "Unable to find record in
            database",
            "Record Not Found",
            JOptionPane.ERROR_MESSAGE);

        } // end if

    } // end try

    catch (SQLException e) {
        JOptionPane.showMessageDialog(this,
            "SQL Exception : " +
            e.getMessage(),
            "SQL ERROR",
            JOptionPane.ERROR_MESSAGE);

        } // end catch

    } // end if

} // end assignmentUpdateForm()

```

```
/**
```

```

* Method languageUpdateForm retrieves and displays a language
* record for a person, which can then be modified or deleted.
* @param none
* @return void
*/

```

```

public void languageUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel idLabel = new JLabel("Enter the Military ID : ");
    final JTextField idField = new JTextField(25);

    JLabel nameLabel = new JLabel("Enter the Foreign Language : ");
}

```

```

final JTextField nameField = new JTextField(25);

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(idLabel);
labelPanel.add(nameLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(idField);
fieldPanel.add(nameField);

getPanel.setLayout(new BorderLayout(getPanel, BorderLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Foreign Language",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM ForeignLanguages WHERE " +
                  "MilitaryID = " + "'" +
                  idField.getText() + "' AND "

                  "Language = " + "'" +
                  nameField.getText() + "'";

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final LanguageForm form = new LanguageForm(
                "LANGUAGE FORM");

            form.militaryIDField.setText(rs.getString(1));

            form.languageField.setSelectedItem(rs.getString
(2));

            form.degreeField.setSelectedItem(rs.getString(3
));

            form.militaryIDField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
ActionListener()
            {

```

```

e)                                     public void actionPerformed(ActionEvent
                                        {
ForeignLanguages" +                    String updateString = "UPDATE
                                        " SET Language = " + "'" +
form.languageField.getSelectedItemAt() + " , " + "Degree = " + "'" +
form.degreeField.getSelectedItemAt() + "'" +
                                        " WHERE MilitaryID = " + "'"
+                                       idField.getText() + "' AND "
+                                       "Language = " + "'" +
                                        nameField.getText() + "'";
                                        updateQuery(updateString);
                                        form.dispose();
                                        }
                                        });

```

```

ActionListener()                       form.deleteButton.addActionListener(new
                                        {
e)                                     public void actionPerformed(ActionEvent
                                        {
+ "'" + +                               String deleteString = "DELETE FROM " +
                                        "ForeignLanguages " +
AND " +                                 "WHERE MilitaryID = "
                                        idField.getText() + "'
                                        "Language = " + "'" +
                                        nameField.getText() +
                                        "'";
                                        updateQuery(deleteString);
                                        form.dispose();
                                        }
                                        });

```

```

ActionListener()                       form.cancelButton.addActionListener(new
                                        {

```



```

        public void actionPerformed(ActionEvent
        e)
        {
            form.dispose();
        }
    });

}

else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);

    } // end if

} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);

    } // end catch

} // end if

} // end languageUpdateForm()

```

```
/**
```

```
* Method eventUpdateForm retrieves and displays an exercise
* event record, which can then be modified or deleted.
```

```
* @param none
* @return void
*/
```

```
public void eventUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel exerciseLabel = new JLabel("Enter the Exercise Name : ");
    final JTextField exerciseField = new JTextField(25);

    JLabel eventLabel = new JLabel("Enter the Event Name : ");

```

```

final JTextField eventField = new JTextField(25);

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(exerciseLabel);
labelPanel.add(eventLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(exerciseField);
fieldPanel.add(eventField);

getPanel.setLayout(new BorderLayout(getPanel, BorderLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Exercise Event",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM Events WHERE " +
                  "ExerciseName = " + "'" +
                  exerciseField.getText() + "'
                  AND " +
                  "EventName = " + "'" +
                  eventField.getText() + "'";

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final EventForm form = new EventForm(
                "EXERCISE/EVENT FORM");

            form.exerciseField.setText(rs.getString(1));
            form.eventField.setText(rs.getString(2));

            form.typeField.setSelectedItem(rs.getString(3))
            ;

            form.numberField.setText(rs.getString(4));
            form.durationField.setText(rs.getString(5));

            form.exerciseField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
            ActionListener()
            {

```

```

public void actionPerformed(ActionEvent
e)
{
    String updateString = "UPDATE
Events SET " +
    "EventName = " + "'" +
    form.eventField.getText() +
    "', " +
    "EventType = " + "'" +
    form.typeField.getSelectedItem() + "', " +
    "NumberOfEvents = " +
    "NumberOfEvents = " +
    form.numberField.getText() +
    "EventDuration = " +
    "EventDuration = " +
    form.durationField.getText()
    " WHERE ExerciseName = " +
    exerciseField.getText() + "'
    AND " +
    "EventName = " + "'" +
    eventField.getText() + "'";

    updateQuery(updateString);

    form.dispose();
});

form.deleteButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM " +
        "Events " +
        "WHERE ExerciseName =
        " + "'" +
        exerciseField.getText() + "' AND " +
        "EventName = " + "'" +
        eventField.getText() +
        "'";

        updateQuery(deleteString);

        form.dispose();
    }
});

```

```

        form.cancelButton.addActionListener(new
ActionListener()
    {
        public void actionPerformed(ActionEvent
e)
        {
            form.dispose();
        }
    });
}
else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);
} // end if
} // end try
catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
} // end catch
} // end if
} // end eventUpdateForm()

```

```

/**
 * Method visitUpdateForm retrieves and displays a port visit
 * record, which can then be modified or deleted.
 * @param none
 * @return void
 */

```

```

public void visitUpdateForm() {

    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

    JLabel exerciseLabel = new JLabel("Enter the Exercise Name : ");

```

```

final JTextField exerciseField = new JTextField(25);

JLabel portLabel = new JLabel("Enter the Port Name : ");
final JTextField portField = new JTextField(25);

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(exerciseLabel);
labelPanel.add(portLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(exerciseField);
fieldPanel.add(portField);

getPanel.setLayout(new BorderLayout(getPanel, BorderLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Port Visit",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM PortVisits WHERE " +
                  "ExerciseName = " + "'" +
                  exerciseField.getText() + "'
AND " +
                  "PortName = " + "'" +
                  portField.getText() + "'";

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final PortVisitForm form = new PortVisitForm(
                "PORT VISIT FORM");

            form.exerciseField.setText(rs.getString(1));
            form.portField.setText(rs.getString(2));
            form.startDateField.setText(rs.getString(3));
            form.endDateField.setText(rs.getString(4));
            form.durationField.setText(rs.getString(5));

            form.exerciseField.setEditable(false);

            form.addButton.setEnabled(false);

            form.updateButton.addActionListener(new
                ActionListener()

```

```

        {
            public void actionPerformed(ActionEvent
            e)
                {
                    String updateString = "UPDATE
                    PortVisits SET " +
                    "PortName = " + "'" +
                    form.portField.getText() +
                    "VisitStartDate = " + "'" +
                    form.startDateField.getText() + "'", " +
                    "VisitEndDate = " + "'" +
                    form.endDateField.getText()
                    + "'", " +
                    "VisitDuration = " +
                    form.durationField.getText()
                    +
                    " WHERE ExerciseName = " +
                    exerciseField.getText() + "'"
                    AND " +
                    "PortName = " + "'" +
                    portField.getText() + "'"";
                    updateQuery(updateString);
                    form.dispose();
                }
        });

        form.deleteButton.addActionListener(new
        ActionListener()
        {
            e)
                {
                    public void actionPerformed(ActionEvent
                    {
                        String deleteString = "DELETE FROM " +
                        "PortVisits " +
                        "WHERE ExerciseName =
                        " + "'" +
                        exerciseField.getText() + "'" AND " +
                        "PortName = " + "'" +
                        portField.getText() +
                        "'";
                        updateQuery(deleteString);
                        form.dispose();
                    }
                }
        });

```

```

        form.cancelButton.addActionListener(new
ActionListener()
    {
        public void actionPerformed(ActionEvent
e)
        {
            form.dispose();
        }
    });
}
else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);
} // end if
} // end try
catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
} // end catch
} // end if
} // end visitUpdateForm()

```

```

/**
 * Method failureUpdateForm retrieves and displays an equipment
 * failure record, which can then be modified or deleted.
 * @param none
 * @return void
 */

```

```

public void failureUpdateForm() {
    JPanel getPanel = new JPanel();
    JPanel labelPanel = new JPanel();
    JPanel fieldPanel = new JPanel();

```

```

JLabel equipmentLabel = new JLabel("Enter the Serial Number : ");
final JTextField equipmentField = new JTextField(25);

JLabel numberLabel = new JLabel("Enter the Failure Number : ");
final JTextField numberField = new JTextField(25);

labelPanel.setLayout(new GridLayout(0, 1));
labelPanel.add(equipmentLabel);
labelPanel.add(numberLabel);

fieldPanel.setLayout(new GridLayout(0, 1));
fieldPanel.add(equipmentField);
fieldPanel.add(numberField);

getPanel.setLayout(new BoxLayout(getPanel, BoxLayout.X_AXIS));
getPanel.add(labelPanel);
getPanel.add(fieldPanel);

String[] optionNames = { "OK", "Cancel" };

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Equipment Failure",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    String query = "SELECT * FROM Failures WHERE " +
                  "SerialNumber = " + "'" +
AND " +
                  equipmentField.getText() + "'
                  "FailureNumber = " +
                  numberField.getText();

    ResultSet rs = selectQuery(query);

    try {
        if (rs != null) {

            rs.next();

            final FailureForm form = new FailureForm(
                "EQUIPMENT FAILURE FORM");

            form.serialField.setText(rs.getString(1));
            form.failureField.setText(rs.getString(2));
            form.descriptionField.setText(rs.getString(3));
            form.diagnosisField.setText(rs.getString(4));
            form.dateField.setText(rs.getString(5));
            form.durationField.setText(rs.getString(6));

            form.serialField.setEditable(false);

            form.addButton.setEnabled(false);
        }
    }
}

```



```

form.updateButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String updateString = "UPDATE
Failures SET " +
        "FailureNumber = " +
        form.failureField.getText()
+ ", " +
        "FailureDescription = " +
        form.descriptionField.getText() + "', " +
        "FailureDiagnosis = " + "'"
+
        form.diagnosisField.getText() + "', " +
        "FailureDate = " + "'" +
        form.dateField.getText() +
        "FailureDuration = " +
        form.durationField.getText()
+
        " WHERE SerialNumber = " +
        equipmentField.getText() +
        "FailureNumber = " +
        numberField.getText();
        updateQuery(updateString);
        form.dispose();
    }
});

```

```

form.deleteButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        String deleteString = "DELETE FROM " +
        "Failures " +
        "WHERE SerialNumber =
" + "'" +
        equipmentField.getText() + "' AND " +
        "FailureNumber = " +
        numberField.getText();
    }
});

```

```

        updateQuery(deleteString);
        form.dispose();
    }
});

form.cancelButton.addActionListener(new
ActionListener()
{
    public void actionPerformed(ActionEvent
e)
    {
        form.dispose();
    }
});

}

else {
    JOptionPane.showMessageDialog(this,
        "Unable to find record in
        database",
        "Record Not Found",
        JOptionPane.ERROR_MESSAGE);

    } // end if

} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);

    } // end catch

} // end if

} // end failureUpdateForm()

```

```

/**
 * Method divisionReport retrieves information about the divisions
 * under each department.
 * @param none
 * @return void
 */

```

```

public void divisionReport() {

    JFrame reportFrame = new JFrame("DIVISION REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

    JLabel departmentLabel = new JLabel("DEPARTMENT",
        SwingConstants.CENTER);
    departmentLabel.setFont(labelFont);

    JLabel divisionLabel = new JLabel("DIVISION",
        SwingConstants.CENTER);
    divisionLabel.setFont(labelFont);

    JLabel officerLabel = new JLabel("OFFICERS",
        SwingConstants.CENTER);
    officerLabel.setFont(labelFont);

    JLabel pettyLabel = new JLabel("PETTY OFFICERS",
        SwingConstants.CENTER);
    pettyLabel.setFont(labelFont);

    JLabel enlistedLabel = new JLabel("ENLISTED",
        SwingConstants.CENTER);
    enlistedLabel.setFont(labelFont);

    labelPanel.add(departmentLabel);
    labelPanel.add(divisionLabel);
    labelPanel.add(officerLabel);
    labelPanel.add(pettyLabel);
    labelPanel.add(enlistedLabel);

    String query = "SELECT * FROM DivisionQuery";

    final ResultSet rs = selectQuery(query);

    if (rs == null) {
        return;
    } // end if

    JPanel textPanel = new JPanel()
    {
        public void paint(Graphics g)
        {
            int yPos = 30;
            String department = " ";

            try {
                while(rs.next()) {
                    g.setFont(textFont);
                    String str = rs.getString(1);

```

```

        if (str.equals(department) == false) {
            department = str;
            yPos += 10;
        g.drawString(str, 10, yPos);
        g.drawString(rs.getString(2), 190,
yPos);
        g.drawString(rs.getString(3), 410,
yPos);
        g.drawString(rs.getString(4), 550,
yPos);
        g.drawString(rs.getString(5), 720,
yPos);
            yPos += 30;
        }
        else {
            g.drawString(rs.getString(2), 190,
yPos);
            g.drawString(rs.getString(3), 410,
yPos);
            g.drawString(rs.getString(4), 550,
yPos);
            g.drawString(rs.getString(5), 720,
yPos);
            yPos += 30;
        } // end if
    } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
    } // end catch
} // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportViewView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

} // end divisionReport()

```

```

/**
 * Method overhaulReport retrieves information about the
 * overhauls of the ship.
 * @param none
 * @return void
 */

public void overhaulReport() {

    JFrame reportFrame = new JFrame("OVERHAUL REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

    JLabel numberLabel = new JLabel("OVERHAUL",
SwingConstants.CENTER);
    numberLabel.setFont(labelFont);

    JLabel startLabel = new JLabel("START DATE",
SwingConstants.CENTER);
    startLabel.setFont(labelFont);

    JLabel endLabel = new JLabel("END DATE",
SwingConstants.CENTER);
    endLabel.setFont(labelFont);

    JLabel shipyardLabel = new JLabel("SHIPYARD",
SwingConstants.CENTER);
    shipyardLabel.setFont(labelFont);

    labelPanel.add(numberLabel);
    labelPanel.add(startLabel);
    labelPanel.add(endLabel);
    labelPanel.add(shipyardLabel);

    String query = "SELECT * FROM OverhaulQuery";

    final ResultSet rs = selectQuery(query);

    if (rs == null) {
        return;
    } // end if

    JPanel textPanel = new JPanel()
{
    public void paint(Graphics g)

```

```

        {
            int yPos = 40;
            try {
                while(rs.next()) {
                    g.setFont(textFont);
                    g.drawString(rs.getString(1), 70, yPos);
                    g.drawString(rs.getString(2), 200, yPos);
                    g.drawString(rs.getString(3), 400, yPos);
                    g.drawString(rs.getString(4), 550, yPos);
                    yPos += 40;
                } // end while
            } // end try

            catch (SQLException e) {
                JOptionPane.showMessageDialog(this,
                    "SQL Exception : " +
                    e.getMessage(),
                    "SQL ERROR",
                    JOptionPane.ERROR_MESSAGE);
            } // end catch
        } // end paint()
};

JScrollPane pane = new JScrollPane();
pane.setViewPortView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

} // end overhaulReport()

/**
 * Method trainingReport retrieves information about the courses
 * taken by the personnel.
 * @param none
 * @return void
 */

public void trainingReport() {

    JFrame reportFrame = new JFrame("TRAINING REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

```

```

        JLabel firstLabel = new JLabel("FIRST NAME",
SwingConstants.CENTER);
        firstLabel.setFont(labelFont);

        JLabel lastLabel = new JLabel("LAST NAME",
SwingConstants.CENTER);
        lastLabel.setFont(labelFont);

        JLabel courseLabel = new JLabel("COURSE NAME",
SwingConstants.CENTER);
        courseLabel.setFont(labelFont);

        JLabel gradeLabel = new JLabel("GRADE",
SwingConstants.CENTER);
        gradeLabel.setFont(labelFont);

        labelPanel.add(firstLabel);
        labelPanel.add(lastLabel);
        labelPanel.add(courseLabel);
        labelPanel.add(gradeLabel);

        String query = "SELECT * FROM TrainingQuery";

        final ResultSet rs = selectQuery(query);

        if (rs == null) {
            return;
        } // end if

        JPanel textPanel = new JPanel()
        {
            public void paint(Graphics g)
            {
                int yPos = 30;
                String militaryID = " ";

                try {
                    while(rs.next()) {
                        g.setFont(textFont);
                        String str = rs.getString(1);

                        if (str.equals(militaryID) == false) {
                            militaryID = str;
                            yPos += 10;
                        }
                        g.drawString(rs.getString(2), 20, yPos);
                        g.drawString(rs.getString(3), 210,
yPos);
                        g.drawString(rs.getString(4), 400,
yPos);
                        g.drawString(rs.getString(5), 700,
yPos);
                    }
                }
            }
        };

```

```

        yPos += 30;
    }
    else {
        g.drawString(rs.getString(4), 400,
yPos);
        g.drawString(rs.getString(5), 700,
yPos);
        yPos += 30;
    } // end if
    } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
    } // end catch
} // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportViewView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

} // end trainingReport()

```

```

/**
 * Method assignmentReport retrieves information about the
 * previous assignments of the personnel.
 * @param none
 * @return void
 */

```

```

public void assignmentReport() {

    JFrame reportFrame = new JFrame("ASSIGNMENT REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

    JLabel firstLabel = new JLabel("FIRST NAME",

```



```

SwingConstants.CENTER);
    firstLabel.setFont(labelFont);

    JLabel lastLabel = new JLabel("LAST NAME",
        SwingConstants.CENTER);
    lastLabel.setFont(labelFont);

    JLabel stationLabel = new JLabel("STATION",
        SwingConstants.CENTER);
    stationLabel.setFont(labelFont);

    JLabel positionLabel = new JLabel("POSITION",
        SwingConstants.CENTER);
    positionLabel.setFont(labelFont);

    labelPanel.add(firstLabel);
    labelPanel.add(lastLabel);
    labelPanel.add(stationLabel);
    labelPanel.add(positionLabel);

    String query = "SELECT * FROM AssignmentQuery";

    final ResultSet rs = selectQuery(query);

    if (rs == null) {
        return;
    } // end if

    JPanel textPanel = new JPanel()
    {
        public void paint(Graphics g)
        {
            int yPos = 30;
            String militaryID = " ";

            try {
                while(rs.next()) {
                    g.setFont(textFont);
                    String str = rs.getString(1);

                    if (str.equals(militaryID) == false) {
                        militaryID = str;
                        yPos += 10;
                    }
                    g.drawString(rs.getString(2), 20, yPos);
                    g.drawString(rs.getString(3), 200,
yPos);
                    g.drawString(rs.getString(4), 380,
yPos);
                    g.drawString(rs.getString(5), 600,
yPos);
                    yPos += 30;
                }
            }
        }
    };

```

```

        }
        else {
            g.drawString(rs.getString(4), 380,
yPos);
            g.drawString(rs.getString(5), 600,
yPos);
            yPos += 30;
        } // end if
    } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
    } // end catch
} // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportViewView(textPanel);

```

```

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

```

```

} // end assignmentReport()

```

```

/**
 * Method languageReport retrieves information about the
 * foreign languages known by the personnel.
 * @param none
 * @return void
 */

```

```

public void languageReport() {

```

```

    JFrame reportFrame = new JFrame("LANGUAGE REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

```

```

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

```

```

    JLabel firstLabel = new JLabel("FIRST NAME",

```

```

SwingConstants.CENTER);
    firstLabel.setFont(labelFont);

    JLabel lastLabel = new JLabel("LAST NAME",
                                   SwingConstants.CENTER);
    lastLabel.setFont(labelFont);

    JLabel languageLabel = new JLabel("LANGUAGE",
                                       SwingConstants.CENTER);
    languageLabel.setFont(labelFont);

    JLabel degreeLabel = new JLabel("DEGREE",
                                     SwingConstants.CENTER);
    degreeLabel.setFont(labelFont);

    labelPanel.add(firstLabel);
    labelPanel.add(lastLabel);
    labelPanel.add(languageLabel);
    labelPanel.add(degreeLabel);

    String query = "SELECT * FROM LanguageQuery";

    final ResultSet rs = selectQuery(query);

    if (rs == null) {
        return;
    } // end if

    JPanel textPanel = new JPanel()
    {
        public void paint(Graphics g)
        {
            int yPos = 30;
            String militaryID = " ";

            try {
                while(rs.next()) {
                    g.setFont(textFont);
                    String str = rs.getString(1);

                    if (str.equals(militaryID) == false) {
                        militaryID = str;
                        yPos += 10;
                    }
                    g.drawString(rs.getString(2), 20, yPos);
                    g.drawString(rs.getString(3), 200,
                                yPos);
                    g.drawString(rs.getString(4), 420,
                                yPos);
                    g.drawString(rs.getString(5), 680,
                                yPos);
                    yPos += 30;
                }
            }
        }
    };

```

```

        else {
            g.drawString(rs.getString(4), 420,
yPos);
            g.drawString(rs.getString(5), 680,
yPos);
            yPos += 30;
        } // end if
    } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
    } // end catch
} // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

```

```

} // end languageReport()

```

```

/**
 * Method eventReport retrieves information about the events
 * executed during the exercises.
 * @param none
 * @return void
 */

```

```

public void eventReport() {

    JFrame reportFrame = new JFrame("EXERCISE/EVENT REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

    JLabel exerciseLabel = new JLabel("EXERCISE NAME",
        SwingConstants.CENTER);
    exerciseLabel.setFont(labelFont);
}

```

```

        JLabel eventLabel = new JLabel("EVENT NAME",
SwingConstants.CENTER);
        eventLabel.setFont(labelFont);

        JLabel typeLabel = new JLabel("EVENT TYPE",
SwingConstants.CENTER);
        typeLabel.setFont(labelFont);

        JLabel durationLabel = new JLabel("DURATION(Hours)",
SwingConstants.CENTER);
        durationLabel.setFont(labelFont);

        labelPanel.add(exerciseLabel);
        labelPanel.add(eventLabel);
        labelPanel.add(typeLabel);
        labelPanel.add(durationLabel);

        String query = "SELECT * FROM EventQuery";

        final ResultSet rs = selectQuery(query);

        if (rs == null) {
            return;
        } // end if

        JPanel textPanel = new JPanel()
    {
        public void paint(Graphics g)
            {
                int yPos = 30;
                String exercise = " ";

                try {
                    while(rs.next()) {
                        g.setFont(textFont);
                        String str = rs.getString(1);

                        if (str.equals(exercise) == false) {
                            exercise = str;
                            yPos += 10;
                        }
                        g.drawString(str, 20, yPos);
                        g.drawString(rs.getString(2), 200,
yPos);
                        g.drawString(rs.getString(3), 400,
yPos);
                        g.drawString(rs.getString(4), 690,
yPos);
                            yPos += 30;
                        }
                    }
                }
            }
        }
    }

```

```

yPos);
yPos);
yPos);
g.drawString(rs.getString(2), 200,
g.drawString(rs.getString(3), 400,
g.drawString(rs.getString(4), 690,
yPos += 30;
    } // end if
  } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);

    } // end catch
} // end paint()
};

JScrollPane pane = new JScrollPane();
pane.setViewportViewView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

} // end eventReport()

```

```

/**
 * Method visitReport retrieves information about the port visits
 * made during the exercises.
 * @param none
 * @return void
 */

```

```

public void visitReport() {

    JFrame reportFrame = new JFrame("PORT VISIT REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

    JLabel exerciseLabel = new JLabel("EXERCISE NAME",

```

```

SwingConstants.CENTER);
exerciseLabel.setFont(labelFont);

JLabel portLabel = new JLabel("PORT NAME",

SwingConstants.CENTER);
portLabel.setFont(labelFont);

JLabel startLabel = new JLabel("START DATE",

SwingConstants.CENTER);
startLabel.setFont(labelFont);

JLabel endLabel = new JLabel("END DATE",

SwingConstants.CENTER);
endLabel.setFont(labelFont);

labelPanel.add(exerciseLabel);
labelPanel.add(portLabel);
labelPanel.add(startLabel);
labelPanel.add(endLabel);

String query = "SELECT * FROM VisitQuery";

final ResultSet rs = selectQuery(query);

if (rs == null) {
    return;
} // end if

JPanel textPanel = new JPanel()
{
    public void paint(Graphics g)
    {
        int yPos = 30;
        String exercise = " ";

        try {
            while(rs.next()) {
                g.setFont(textFont);
                String str = rs.getString(1);

                if (str.equals(exercise) == false) {
                    exercise = str;
                    yPos += 10;
                }
                g.drawString(str, 20, yPos);
                g.drawString(rs.getString(2), 230,
yPos);
                g.drawString(rs.getString(3), 430,
yPos);
                g.drawString(rs.getString(4), 650,
yPos);
                yPos += 30;
            }
        }
    }
};

```

```

        }
        else {
yPos);          g.drawString(rs.getString(2), 230,
yPos);          g.drawString(rs.getString(3), 430,
yPos);          g.drawString(rs.getString(4), 650,
                yPos += 30;
                } // end if
            } // end while
        } // end try

        catch (SQLException e) {
            JOptionPane.showMessageDialog(this,
                "SQL Exception : " +
                e.getMessage(),
                "SQL ERROR",
                JOptionPane.ERROR_MESSAGE);
        } // end catch
    } // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportViewView(textPanel);

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

```

```

} // end visitReport()

```

```

/**

```

```

 * Method failureReport retrieves information about the failures
 * of equipment.
 * @param none
 * @return void
 */

```

```

public void failureReport() {

```

```

    JFrame reportFrame = new JFrame("EQUIPMENT FAILURE REPORT");
    reportFrame.setSize(800, 600);
    reportFrame.setBackground(Color.lightGray);

```

```

    JPanel labelPanel = new JPanel();
    labelPanel.setSize(800, 100);
    labelPanel.setLayout(new GridLayout(1, 0));

```



```

JLabel equipmentLabel = new JLabel("EQUIPMENT NAME",
    SwingConstants.CENTER);
equipmentLabel.setFont(labelFont);

JLabel typeLabel = new JLabel("EQUIPMENT TYPE",
    SwingConstants.CENTER);
typeLabel.setFont(labelFont);

JLabel descriptionLabel = new JLabel("FAILURE",
    SwingConstants.CENTER);
descriptionLabel.setFont(labelFont);

JLabel durationLabel = new JLabel("DURATION(Hours)",
    SwingConstants.CENTER);
durationLabel.setFont(labelFont);

labelPanel.add(equipmentLabel);
labelPanel.add(typeLabel);
labelPanel.add(descriptionLabel);
labelPanel.add(durationLabel);

String query = "SELECT * FROM FailureQuery";

final ResultSet rs = selectQuery(query);

if (rs == null) {
    return;
} // end if

JPanel textPanel = new JPanel()
{
    public void paint(Graphics g)
    {
        int yPos = 30;
        String equipment = " ";

        try {
            while(rs.next()) {
                g.setFont(textFont);
                String str = rs.getString(1);

                if (str.equals(equipment) == false) {
                    equipment = str;
                    yPos += 10;
                }
                g.drawString(str, 20, yPos);
                g.drawString(rs.getString(2), 200,
yPos);
                g.drawString(rs.getString(3), 420,
yPos);
                g.drawString(rs.getString(4), 700,
yPos);
            }
        }
    }
}

```

```

        yPos += 30;
    }
    else {
        g.drawString(rs.getString(3), 420,
yPos);
        g.drawString(rs.getString(4), 700,
yPos);
        yPos += 30;
    } // end if
    } // end while
} // end try

catch (SQLException e) {
    JOptionPane.showMessageDialog(this,
        "SQL Exception : " +
        e.getMessage(),
        "SQL ERROR",
        JOptionPane.ERROR_MESSAGE);
    } // end catch
} // end paint()
};

```

```

JScrollPane pane = new JScrollPane();
pane.setViewportView(textPanel);

```

```

reportFrame.getContentPane().setLayout(new BorderLayout());
reportFrame.getContentPane().add(labelPanel, BorderLayout.NORTH);
reportFrame.getContentPane().add(pane, BorderLayout.CENTER);
reportFrame.show();

```

```

} // end failureReport()

```

```

/**
 * Method courseToTakeQuery retrieves information about the courses
 * that should be taken by the personnel.
 * @param none
 * @return void
 */

```

```

public void courseToTakeQuery() {

```

```

    JLabel nameLabel = new JLabel("Enter the Last Name : ");
    JTextField nameField = new JTextField(25);

```

```

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

```

```

    String[] optionNames = { "OK", "Cancel" };

```

```

if (JOptionPane.showOptionDialog(this, getPanel,
                                "Select Personnel",
                                JOptionPane.YES_NO_CANCEL_OPTION,
                                JOptionPane.QUESTION_MESSAGE,
                                null, optionNames,
                                optionNames[0]) == 0) {

    JDBCAdapter dbadapter;
    dbadapter = new JDBCAdapter(connectionPanel.getURL(),

                                connectionPanel.getDriver(),

                                connectionPanel.getUserID(),

                                connectionPanel.getPassword());

    dbadapter.connect();

    String query = "SELECT * FROM CourseToTakeQuery WHERE " +
                   "LastName = " + "'" +
                   nameField.getText() + "'";

    dbadapter.executeQuery(query);

    final JTable table = new JTable(dbadapter);

    JScrollPane scrollPane = new JScrollPane(table);

    JPanel controlPanel = new JPanel();

    JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
    controlPanel.add(cellSpacingLabel);

    JSlider cellSpacingSlider = new JSlider(

    JSlider.HORIZONTAL, 0, 10, 1);
    cellSpacingSlider.getAccessibleContext().

    setAccessibleName("Inter-Cell Spacing");
    cellSpacingLabel.setLabelFor(cellSpacingSlider);
    controlPanel.add(cellSpacingSlider);

    cellSpacingSlider.addChangeListener(new ChangeListener()
    {
        public void stateChanged(ChangeEvent e)
        {
            int spacing = ((JSlider) e.getSource()).getValue();
            table.setInterCellSpacing(new
            Dimension(spacing, spacing));
            table.repaint();
        }
    });

    JLabel rowHeightLabel = new JLabel("Row Height");
    controlPanel.add(rowHeightLabel);

```

```

        JSlider rowHeightSlider = new JSlider(
            JSlider.HORIZONTAL, 5, 100, 20);
        rowHeightSlider.getAccessibleContext().
            setAccessibleName("Row Height");
        rowHeightLabel.setLabelFor(rowHeightSlider);
        controlPanel.add(rowHeightSlider);

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("COURSES TO TAKE QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER
    );

        frame.getContentPane().add(controlPanel,
            BorderLayout.NORTH);
        frame.show();

    } // end if

} // end courseToTakeQuery()

```

```

/**
 * Method courseTakenQuery retrieves information about the courses
 * that were taken by the personnel.
 * @param none
 * @return void
 */

```

```

public void courseTakenQuery() {

    JLabel nameLabel = new JLabel("Enter the Last Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,

```

```

        "Select Personnel",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,
        optionNames[0]) == 0) {

JDBCAdapter dbadapter;
dbadapter = new JDBCAdapter(connectionPanel.getURL(),

        connectionPanel.getDriver(),

        connectionPanel.getUserID(),

        connectionPanel.getPassword());

dbadapter.connect();

String query = "SELECT * FROM CourseTakenQuery WHERE " +
                "LastName = " + "'" +
                nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(

JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().

setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new
            Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

```

```

        JSlider rowHeightSlider = new JSlider(
            JSlider.HORIZONTAL, 5, 100, 20);
        rowHeightSlider.getAccessibleContext().
            setAccessibleName("Row Height");
        rowHeightLabel.setLabelFor(rowHeightSlider);
        controlPanel.add(rowHeightSlider);

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("COURSES TAKEN QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
    };

    frame.getContentPane().add(controlPanel,
        BorderLayout.NORTH);
    frame.show();

} // end if

} // end courseTakenQuery()

```

```

/**
 * Method assignmentQuery retrieves information about the previous
 * assignments of the personnel.
 * @param none
 * @return void
 */

```

```

public void assignmentQuery() {

    JLabel nameLabel = new JLabel("Enter the Last Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Personnel",

```

```

        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,
        optionNames[0]) == 0) {

JDBCAdapter dbadapter;
dbadapter = new JDBCAdapter(connectionPanel.getURL(),

        connectionPanel.getDriver(),

        connectionPanel.getUserID(),

        connectionPanel.getPassword());
dbadapter.connect();

String query = "SELECT * FROM PreviousQuery WHERE " +
                "LastName = " + "'" +
                nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(

JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().

setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new
        Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(

```

```

        JSlider.HORIZONTAL, 5, 100, 20);
        rowHeightSlider.getAccessibleContext().
            setAccessibleName("Row Height");
        rowHeightLabel.setLabelFor(rowHeightSlider);
        controlPanel.add(rowHeightSlider);

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("PREVIOUS ASSIGNMENTS QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);
    };

    frame.getContentPane().add(controlPanel,
        BorderLayout.NORTH);
    frame.show();

} // end if

} // end assignmentQuery()

```

```

/**
 * Method languageQuery retrieves information about the foreign
 * languages known by the personnel.
 * @param none
 * @return void
 */

public void languageQuery() {

    JLabel nameLabel = new JLabel("Enter the Last Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Personnel",
        JOptionPane.YES_NO_CANCEL_OPTION,

```



```

        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,
        optionNames[0]) == 0) {

JDBCAdapter dbadapter;
dbadapter = new JDBCAdapter(connectionPanel.getUrl(),

        connectionPanel.getDriver(),

        connectionPanel.getUserID(),

        connectionPanel.getPassword());
dbadapter.connect();

String query = "SELECT * FROM ForeignQuery WHERE " +
                "LastName = " + "'" +
                nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(
        JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.setAccessibleContext().
setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new
        Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(
        JSlider.HORIZONTAL, 5, 100, 20);

```

```

        rowHeightSlider.getAccessibleContext().
            setAccessibleName("Row Height");
        rowHeightLabel.setLabelFor(rowHeightSlider);
        controlPanel.add(rowHeightSlider);

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("FOREIGN LANGUAGE QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);

        frame.getContentPane().add(controlPanel,
            BorderLayout.NORTH);
        frame.show();

    } // end if

} // end languageQuery()

/**
 * Method eventQuery retrieves information about the events
 * executed during the exercises.
 * @param none
 * @return void
 */

public void eventQuery() {

    JLabel nameLabel = new JLabel("Enter the Exercise Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Exercise",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,

```

```

        optionNames[0]) == 0) {

JDBCAdapter dbadapter;
dbadapter = new JDBCAdapter(connectionPanel.getURL(),

        connectionPanel.getDriver(),

        connectionPanel.getUserID(),

        connectionPanel.getPassword());
dbadapter.connect();

String query = "SELECT * FROM EventQuery WHERE " +
               "ExerciseName = " + "'" +
               nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(

JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().

setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new
        Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(

JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");

```

```

        rowHeightLabel.setLabelFor(rowHeightSlider);
        controlPanel.add(rowHeightSlider);

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("EXERCISE/EVENT QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);

        frame.getContentPane().add(controlPanel,
        BorderLayout.NORTH);
        frame.show();

    } // end if
} // end eventQuery()

```

```

/**
 * Method visitQuery retrieves information about the port visits
 * made during the exercises.
 * @param none
 * @return void
 */
public void visitQuery() {

    JLabel nameLabel = new JLabel("Enter the Exercise Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Exercise",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,
        optionNames[0]) == 0) {

```

```

JDBCAdapter dbadapter;
dbadapter = new JDBCAdapter(connectionPanel.getURL(),

                           connectionPanel.getDriver(),

                           connectionPanel.getUserID(),

                           connectionPanel.getPassword());
dbadapter.connect();

String query = "SELECT * FROM VisitQuery WHERE " +
              "ExerciseName = " + "'" +
              nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(

JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().

setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setIntercellSpacing(new
            Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(

JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
    setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

```

```

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("PORT VISIT QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);

        frame.getContentPane().add(controlPanel,
        BorderLayout.NORTH);
        frame.show();

    } // end if

} // end visitQuery()

/**
 * Method failureQuery retrieves information about the failures
 * of equipment.
 * @param none
 * @return void
 */

public void failureQuery() {

    JLabel nameLabel = new JLabel("Enter the Equipment Name : ");
    JTextField nameField = new JTextField(25);

    JPanel getPanel = new JPanel();
    getPanel.add(nameLabel);
    getPanel.add(nameField);

    String[] optionNames = { "OK", "Cancel" };

    if (JOptionPane.showOptionDialog(this, getPanel,
        "Select Equipment",
        JOptionPane.YES_NO_CANCEL_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null, optionNames,
        optionNames[0]) == 0) {

        JDBCAdapter dbadapter;
        dbadapter = new JDBCAdapter(connectionPanel.getURL(),

```

```

        connectionPanel.getDriver(),
        connectionPanel.getUserID(),
        connectionPanel.getPassword());
dbadapter.connect();

String query = "SELECT * FROM EquipmentFailureQuery WHERE "
        + "EquipmentName = " + "'" +
        nameField.getText() + "'";

dbadapter.executeQuery(query);

final JTable table = new JTable(dbadapter);

JScrollPane scrollPane = new JScrollPane(table);

JPanel controlPanel = new JPanel();

JLabel cellSpacingLabel = new JLabel("Inter-Cell Spacing");
controlPanel.add(cellSpacingLabel);

JSlider cellSpacingSlider = new JSlider(
        JSlider.HORIZONTAL, 0, 10, 1);
cellSpacingSlider.getAccessibleContext().
        setAccessibleName("Inter-Cell Spacing");
cellSpacingLabel.setLabelFor(cellSpacingSlider);
controlPanel.add(cellSpacingSlider);

cellSpacingSlider.addChangeListener(new ChangeListener()
{
    public void stateChanged(ChangeEvent e)
    {
        int spacing = ((JSlider) e.getSource()).getValue();
        table.setInterCellSpacing(new
            Dimension(spacing, spacing));
        table.repaint();
    }
});

JLabel rowHeightLabel = new JLabel("Row Height");
controlPanel.add(rowHeightLabel);

JSlider rowHeightSlider = new JSlider(
        JSlider.HORIZONTAL, 5, 100, 20);
rowHeightSlider.getAccessibleContext().
        setAccessibleName("Row Height");
rowHeightLabel.setLabelFor(rowHeightSlider);
controlPanel.add(rowHeightSlider);

```

```

        rowHeightSlider.addChangeListener(new ChangeListener()
        {
            public void stateChanged(ChangeEvent e)
            {
                int height = ((JSlider) e.getSource()).getValue();
                table.setRowHeight(height);
                table.repaint();
            }
        });

        JFrame frame = new JFrame("EQUIPMENT FAILURE QUERY");
        frame.setSize(800, 600);
        frame.setBackground(Color.lightGray);
        frame.getContentPane().setLayout(new BorderLayout());
        frame.getContentPane().add(scrollPane, BorderLayout.CENTER);

        frame.getContentPane().add(controlPanel,
        BorderLayout.NORTH);
        frame.show();

    } // end if
} // end failureQuery()

```

```

/**
 * Method showQueryWindow creates and displays a window for writing
 * SQL queries and fetching query results from the database.
 * @param none
 * @return void
 */

```

```

public void showQueryWindow() {

    Color labelColor = new Color(144, 216, 234);
    Font labelFont = new Font("Serif", Font.BOLD, 16);
    Font areaFont = new Font("Serif", Font.PLAIN, 14);

    if (queryFrame == null) {

        // Create the query labels
        selectLabel = new JLabel("SELECT");
        selectLabel.setBackground(labelColor);
        selectLabel.setFont(labelFont);

        fromLabel = new JLabel("FROM");
        fromLabel.setBackground(labelColor);
        fromLabel.setFont(labelFont);

        whereLabel = new JLabel("WHERE");
        whereLabel.setBackground(labelColor);
        whereLabel.setFont(labelFont);
    }
}

```



```

groupLabel = new JLabel("GROUP BY");
groupLabel.setBackground(labelColor);
groupLabel.setFont(labelFont);

havingLabel = new JLabel("HAVING");
havingLabel.setBackground(labelColor);
havingLabel.setFont(labelFont);

orderLabel = new JLabel("ORDER BY");
orderLabel.setBackground(labelColor);
orderLabel.setFont(labelFont);

// Create the query text areas
selectArea = new JTextArea(" ", 2, 30);
selectArea.setFont(areaFont);
fromArea = new JTextArea(" ", 2, 30);
fromArea.setFont(areaFont);
whereArea = new JTextArea(" ", 2, 30);
whereArea.setFont(areaFont);
groupArea = new JTextArea(" ", 2, 30);
groupArea.setFont(areaFont);
havingArea = new JTextArea(" ", 2, 30);
havingArea.setFont(areaFont);
orderArea = new JTextArea(" ", 2, 30);
orderArea.setFont(areaFont);

fetchButton = new JButton("RUN QUERY");
fetchButton.setFont(labelFont);
fetchButton.setBackground(labelColor);

leftPanel = new JPanel();
leftPanel.setLayout(new GridLayout(0, 1));
leftPanel.setSize(200, 600);
leftPanel.setLocation(0, 0);

leftPanel.add(selectLabel);
leftPanel.add(selectArea);
leftPanel.add(fromLabel);
leftPanel.add(fromArea);
leftPanel.add(whereLabel);
leftPanel.add(whereArea);
leftPanel.add(groupLabel);
leftPanel.add(groupArea);
leftPanel.add(havingLabel);
leftPanel.add(havingArea);
leftPanel.add(orderLabel);
leftPanel.add(orderArea);
leftPanel.add(fetchButton);

fetchButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        String query = "SELECT " + selectArea.getText()

```

```

        " FROM " + fromArea.getText();
String whereText = whereArea.getText();
String groupText = groupArea.getText();
String havingText = havingArea.getText();
String orderText = orderArea.getText();
String emptyString = " ";

        if (whereText.compareToIgnoreCase(emptyString)
!= 0) {
            whereText);
                query = query.concat(" WHERE " +
            }

        if (groupText.compareToIgnoreCase(emptyString) !=
0) {
            groupText);
                query = query.concat(" GROUP BY " +
            }

        if (havingText.compareToIgnoreCase(emptyString)
!= 0) {
            havingText);
                query = query.concat(" HAVING " +
            }

        if (orderText.compareToIgnoreCase(emptyString)
!= 0) {
            orderText);
                query = query.concat(" ORDER BY " +
            }

        dataBase.executeQuery(query);

    }
});

// Create the table scrollpane
dataBase = new JDBCAdapter(connectionPanel.getURL(),
                           connectionPanel.getDriver(),
                           connectionPanel.getUserID(),
                           connectionPanel.getPassword());

dataBase.connect();

JTable table = new JTable(dataBase);

tableAggregate = new JScrollPane(table);
tableAggregate.setBorder(new BevelBorder(BevelBorder.LOWERED));
tableAggregate.setSize(600, 600);
tableAggregate.setLocation(200, 0);

// Add all components to the query panel
queryPanel = new JPanel();
queryPanel.setLayout(null);
queryPanel.add(leftPanel);

```

```

        queryPanel.add(tableAggregate);

        // Create a frame and put the queryPanel on it
        queryFrame = new JFrame("QUERY WINDOW");
        queryFrame.setSize(800, 600);
        queryFrame.setBackground(Color.lightGray);
        queryFrame.getContentPane().add(queryPanel);

        queryFrame.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                queryFrame.removeNotify();
            }
        });

        queryFrame.show();
    }

    // A queryFrame already exists, just make it visible
    else {
        queryFrame.setVisible(true);
    } // end if
} // end showQueryWindow()

/**
 * Method updateQuery executes an update operation.
 * @param none
 * @return void
 */
public void updateQuery (String query) {
    try {
        Class.forName(connectionPanel.getDriver());
    }

    catch (ClassNotFoundException cnf) {
        System.err.println("Can not find database driver classes");
        System.err.println(cnf);
    }

    try {
        Connection connection = DriverManager.getConnection(
            connectionPanel.getURL(),
            connectionPanel.getUserID(),

```

```

connectionPanel.getPassword());

        Statement statement = connection.createStatement();
        statement.executeUpdate(query);
        statement.close();
        connection.close();
    }

    catch (SQLException e) {
        JOptionPane.showMessageDialog(this,
            "SQL Exception : " +
            e.getMessage(),
            "SQL ERROR",
            JOptionPane.ERROR_MESSAGE);
    }

} // end updateQuery()

```

```

/**
 * Method selectQuery executes a query and returns the result.
 * @param none
 * @return void
 */

public ResultSet selectQuery (String query) {

    try {
        Class.forName(connectionPanel.getDriver());
    }

    catch (ClassNotFoundException cnf) {
        System.err.println("Can not find database driver classes");
        System.err.println(cnf);
        return null;
    }

    try {
        Connection connection = DriverManager.getConnection(
            connectionPanel.getURL(),
            connectionPanel.getUserID(),
            connectionPanel.getPassword());

        Statement statement = connection.createStatement();

        ResultSet rs = statement.executeQuery(query);

        return rs;
    }
}

```

```
        catch (SQLException e) {
            JOptionPane.showMessageDialog(this,
                "SQL Exception : " +
                e.getMessage(),
                "SQL ERROR",
                JOptionPane.ERROR_MESSAGE);

            return null;
        }
    } // end selectQuery()

} // end class POETApplication
```

```

import javax.swing.*;
import java.awt.*;

/**
 * The ConnectionPanel class inherits from the JPanel class and
 * implements a graphical user interface for the connections to
 * the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class ConnectionPanel extends JPanel {

    JLabel        userNameLabel;
    JTextField    userNameField;
    JLabel        passwordLabel;
    JPasswordField passwordField;
    JPanel        namePanel;
    JPanel        fieldPanel;
    String        server;
    String        driver;

    public ConnectionPanel() {

        server = "jdbc:odbc:POETDB";
        driver = "sun.jdbc.odbc.JdbcOdbcDriver";

        userNameLabel = new JLabel("USER NAME  ", JLabel.RIGHT);
        userNameField = new JTextField(15);

        passwordLabel = new JLabel("PASSWORD  ", JLabel.RIGHT);
        passwordField = new JPasswordField(15);

        namePanel = new JPanel();
        namePanel.setLayout( new GridLayout( 0, 1 ) );
        namePanel.add( userNameLabel );
        namePanel.add( passwordLabel );

        fieldPanel = new JPanel();
        fieldPanel.setLayout( new GridLayout( 0, 1 ) );
        fieldPanel.add( userNameField );
        fieldPanel.add( passwordField );

        setLayout( new BorderLayout( this, BorderLayout.X_AXIS ) );

        add( namePanel );
        add( fieldPanel );

    } // end ConnectionPanel()

```

```
public String getUserID() {  
    return ( userNameField.getText() );  
} // end getUserID()  
  
public String getPassword() {  
    char[] array = passwordField.getPassword();  
    return ( String.valueOf(array) );  
} // end getPassword()  
  
public String getURL() {  
    return ( server );  
} // end getURL()  
  
public String getDriver() {  
    return ( driver );  
} // end getDriver()  
  
} // end class ConnectionPanel
```

```

import java.sql.*;
import java.util.Vector;
import javax.swing.JOptionPane;
import javax.swing.table.AbstractTableModel;
import javax.swing.event.TableModelEvent;

/**
 * The JDBCAdapter class inherits from AbstractTableModel class and
 * provides the TableModel implementation for retrieving the query
 * results from the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class JDBCAdapter extends AbstractTableModel {

    Connection connection;
    Statement statement;
    String url, driver, userid, password;
    ResultSet queryResult;
    ResultSetMetaData metaData;
    String[] columnNames = {};
    Vector rows = new Vector();

    public JDBCAdapter(String URL, String driverName,
                      String user, String passwd) {

        url = URL;
        driver = driverName;
        userid = user;
        password = passwd;
    } // end JDBCAdapter()

    public void connect() {

        try {
            Class.forName(driver);
            connection = DriverManager.getConnection( url,
                                                    userid, password );
            statement = connection.createStatement();
        }

        catch (ClassNotFoundException ex) {
            System.err.println("Cannot find database driver classes");
            System.err.println(ex);
        }
    }

```



```

catch (SQLException ex) {
    System.err.println("Cannot connect to this database.");
    System.err.println(ex);
}

} // end connect()

public boolean isConnected() {
    if (connection == null) {
        return false;
    }
    else {
        return true;
    } // end if
} // end isConnected()

public void executeQuery(String query) throws NullPointerException {
    if( connection == null || statement == null ) {
        System.err.println( "Unable to execute query. " +
            "No connection exists" );
    }
    else {
        try {
            queryResult = statement.executeQuery(query);
            metaData = queryResult.getMetaData();
        }
        catch ( SQLException ex ) {
            JOptionPane.showMessageDialog( null,
                "SQLException: " +
                    ex.getMessage(),
                    "SQL Error",
                    JOptionPane.ERROR_MESSAGE );
        }
    } // end if

    try {

```

```

        int numberOfColumns = metaData.getColumnCount();
        columnNames = new String[ numberOfColumns ];
        for ( int column = 0; column < numberOfColumns;
column++ ) {
            metaData.getColumnLabel(
                columnNames[ column ] =
                column + 1 );
        } // end for
        rows = new Vector();
        while ( queryResult.next() ) {
            Vector newRow = new Vector();
            for ( int i = 1; i <= getColumnCount(); i++ )
            {
                newRow.addElement(
queryResult.getObject( i ) );
            } // end for
            rows.addElement( newRow );
        } // end while
        // Tell the listeners a new table has arrived.
        fireTableChanged( null );
    }
    catch ( SQLException ex ) {
        System.err.println( ex );
    }
} // end executeQuery()

public void close() throws SQLException {
    queryResult.close();
    statement.close();
    connection.close();
} // end close()

protected void finalize() throws Throwable {

```

```

        close();
        super.finalize();
    } // end of finalize()

    public String getColumnName(int column) {
        if (columnNames[column] != null) {
            return columnNames[column];
        }
        else {
            return "";
        } // end if
    } // end getColumnName()

    public Class getColumnClass(int column) {
        int type;
        try {
            type = metaData.getColumnType(column+1);
        }
        catch (SQLException e) {
            return super.getColumnClass(column);
        }

        switch(type) {
        case Types.CHAR:
        case Types.VARCHAR:
        case Types.LONGVARCHAR:
            return String.class;

        case Types.BIT:
            return Boolean.class;

        case Types.TINYINT:
        case Types.SMALLINT:
        case Types.INTEGER:
            return Integer.class;

        case Types.BIGINT:
            return Long.class;

        case Types.FLOAT:
        case Types.DOUBLE:
            return Double.class;

        case Types.DATE:
            return java.sql.Date.class;

        default:

```

```

        return Object.class;
    }

} // end getColumnClass()

public boolean isCellEditable(int row, int column) {
    try {
        return metaData.isWritable(column+1);
    }
    catch (SQLException e) {
        return false;
    }
} // end isCellEditable()

public int getColumnCount() {
    return columnNames.length;
} // end getColumnCount()

public int getRowCount() {
    return rows.size();
} // end getRowCount()

public Object getValueAt(int aRow, int aColumn) {
    Vector row = (Vector)rows.elementAt(aRow);
    return row.elementAt(aColumn);
} // end getValueAt()

public String dbRepresentation(int column, Object value) {
    int type;

    if (value == null) {
        return "null";
    }

    try {
        type = metaData.getColumnType(column+1);
    }

```

```

catch (SQLException e) {
    return value.toString();
}

switch(type) {
    case Types.INTEGER:
    case Types.DOUBLE:
    case Types.FLOAT:
        return value.toString();

    case Types.BIT:
        return ((Boolean)value).booleanValue() ? "1" : "0";

    case Types.DATE:
        return value.toString();

    default:
        return "\"" + value.toString() + "\"";
} // end switch

} // end dbRepresentation()

public void setValueAt(Object value, int row, int column) {
    try {
        String tableName = metaData.getTableName(column+1);

        if (tableName == null) {
            System.out.println("Table name returned null.");
        }

        String columnName = getColumnName(column);
        String query =
            "update " + tableName +
            " set " + columnName + " = " +
                dbRepresentation(column, value) +
            " where ";

        // We don't have a model of the schema so we don't know the
        // primary keys or which columns to lock on. To demonstrate
        // that editing is possible, we'll just lock on everything.
        for(int col = 0; col < getColumnCount(); col++) {
            String colName = getColumnName(col);
            if (colName.equals("")) {
                continue;
            }
            if (col != 0) {
                query = query + " and ";
            }
            query = query + colName + " = " +
                dbRepresentation(col, getValueAt(row, col));
        }
    }
}

```

```
        System.out.println(query);
        System.out.println("Not sending update to database");
        statement.executeQuery(query);
    }

    catch (SQLException e) {
        e.printStackTrace();
        System.err.println("Update failed");
    }

    Vector dataRow = (Vector)rows.elementAt(row);
    dataRow.setElementAt(value, column);

} // end setValueAt()

} // end class JDBCAdapter
```

```
import javax.swing.*;
import java.awt.*;
```

```
/**
 * The AssignmentForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Assignments Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */
```

```
public class AssignmentForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField militaryIDField;
    JLabel militaryIDLabel;

    JTextField numberField;
    JLabel numberLabel;

    JTextField stationField;
    JLabel stationLabel;

    JTextField positionField;
    JLabel positionLabel;

    JTextField durationField;
    JLabel durationLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    public AssignmentForm(String title) {

        super(title);

        militaryIDField = new JTextField(25);
        militaryIDLabel = new JLabel(" Personnel Military ID : ");

        numberField = new JTextField(25);
        numberLabel = new JLabel(" Assignment Number : ");

        stationField = new JTextField(25);
        stationLabel = new JLabel(" Station Name : ");
```

```

positionField = new JTextField(25);
positionLabel = new JLabel(" Position Name : ");

durationField = new JTextField(25);
durationLabel = new JLabel(" Duration (Years) : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(militaryIDLabel);
namePanel.add(numberLabel);
namePanel.add(stationLabel);
namePanel.add(positionLabel);
namePanel.add(durationLabel);

fieldPanel.add(militaryIDField);
fieldPanel.add(numberField);
fieldPanel.add(stationField);
fieldPanel.add(positionField);
fieldPanel.add(durationField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

this.setSize(800, 600);
this.getContentPane().setLayout(new BorderLayout());
this.getContentPane().add(box, BorderLayout.CENTER);
this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
this.show();

} // end AssignmentForm()

} // end class AssignmentForm

```



```

import javax.swing.*;
import java.awt.*;

/**
 * The CourseTakenForm class inherits from the JFrame class and
 * provides a graphical representation, which consists of labels,
 * fields, and combo boxes, for the CoursesTaken Table in the
 * POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class CourseTakenForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField militaryIDField;
    JLabel militaryIDLabel;

    JTextField courseField;
    JLabel courseLabel;

    JTextField startDateField;
    JLabel startDateLabel;

    JTextField endDateField;
    JLabel endDateLabel;

    JTextField gradeField;
    JLabel gradeLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    public CourseTakenForm(String title) {

        super(title);

        militaryIDField = new JTextField(25);
        militaryIDLabel = new JLabel(" Personnel Military ID : ");

        courseField = new JTextField(25);
        courseLabel = new JLabel(" Course Name : ");

        startDateField = new JTextField(25);
        startDateLabel = new JLabel(" Course Start Date : ");
    }

```

```

        endDateField = new JTextField(25);
        endDateLabel = new JLabel(" Course End Date : ");

        gradeField = new JTextField(25);
        gradeLabel = new JLabel(" Course Grade (1..100) : ");

        namePanel = new JPanel();
        fieldPanel = new JPanel();

        namePanel.setLayout(new GridLayout(0, 1));
        fieldPanel.setLayout(new GridLayout(0, 1));

        namePanel.add(militaryIDLabel);
        namePanel.add(courseLabel);
        namePanel.add(startDateLabel);
        namePanel.add(endDateLabel);
        namePanel.add(gradeLabel);

        fieldPanel.add(militaryIDField);
        fieldPanel.add(courseField);
        fieldPanel.add(startDateField);
        fieldPanel.add(endDateField);
        fieldPanel.add(gradeField);

        box = new Box(BoxLayout.X_AXIS);
        box.add(namePanel);
        box.add(fieldPanel);

        addButton = new JButton("ADD RECORD");
        addButton.setBackground(buttonColor);

        deleteButton = new JButton("DELETE RECORD");
        deleteButton.setBackground(buttonColor);

        updateButton = new JButton("UPDATE RECORD");
        updateButton.setBackground(buttonColor);

        cancelButton = new JButton("CANCEL");
        cancelButton.setBackground(buttonColor);

        buttonPanel = new JPanel();
        buttonPanel.add(addButton);
        buttonPanel.add(deleteButton);
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end CourseTakenForm()

} // end class CourseTakenForm

```

```

import javax.swing.*;
import java.awt.*;

/**
 * The CourseToTakeForm class inherits from the JFrame class and
 * provides a graphical representation, which consists of labels,
 * fields, and combo boxes, for the CoursesToTake Table in the
 * POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class CourseToTakeForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField militaryIDField;
    JLabel militaryIDLabel;

    JTextField courseField;
    JLabel courseLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    public CourseToTakeForm(String title) {

        super(title);

        militaryIDField = new JTextField(25);
        militaryIDLabel = new JLabel(" Personnel Military ID : ");

        courseField = new JTextField(25);
        courseLabel = new JLabel(" Course Name : ");

        namePanel = new JPanel();
        fieldPanel = new JPanel();

        namePanel.setLayout(new GridLayout(0, 1));
        fieldPanel.setLayout(new GridLayout(0, 1));

        namePanel.add(militaryIDLabel);
        namePanel.add(courseLabel);

```

```

        fieldPanel.add(militaryIDField);
        fieldPanel.add(courseField);

        box = new Box(BoxLayout.X_AXIS);
        box.add(namePanel);
        box.add(fieldPanel);

        addButton = new JButton("ADD RECORD");
        addButton.setBackground(buttonColor);

        deleteButton = new JButton("DELETE RECORD");
        deleteButton.setBackground(buttonColor);

        updateButton = new JButton("UPDATE RECORD");
        updateButton.setBackground(buttonColor);

        cancelButton = new JButton("CANCEL");
        cancelButton.setBackground(buttonColor);

        buttonPanel = new JPanel();
        buttonPanel.add(addButton);
        buttonPanel.add(deleteButton);
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end CourseToTakeForm()

} // end class CourseToTakeForm

```

```

import javax.swing.*;
import java.awt.*;

/**
 * The EquipmentForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Equipment Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class EquipmentForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField serialNumberField;
    JLabel serialNumberLabel;

    JTextField stockNumberField;
    JLabel stockNumberLabel;

    JTextField nameField;
    JLabel nameLabel;

    JComboBox typeField;
    JLabel typeLabel;

    JTextField dateField;
    JLabel dateLabel;

    JTextField manufacturerField;
    JLabel manufacturerLabel;

    JTextField modelField;
    JLabel modelLabel;

    JTextField locationField;
    JLabel locationLabel;

    JTextField runtimeField;
    JLabel runtimeLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

```

```

static String[] type = { "COMMUNICATIONS", "RADAR",
                        "ELECTRONIC WARFARE", "NAVIGATION",
                        "WEAPONS", "ENGINE" };

public EquipmentForm(String title) {

    super(title);

    serialNumberField = new JTextField(25);
    serialNumberLabel = new JLabel(" Serial Number : ");

    stockNumberField = new JTextField(25);
    stockNumberLabel = new JLabel(" Stock Number : ");

    nameField = new JTextField(25);
    nameLabel = new JLabel(" Equipment Name : ");

    typeField = new JComboBox(type);
    typeField.setEditable(false);
    typeLabel = new JLabel(" Equipment Type : ");

    dateField = new JTextField(25);
    dateLabel = new JLabel(" Production Date : ");

    manufacturerField = new JTextField(25);
    manufacturerLabel = new JLabel(" Manufacturer : ");

    modelField = new JTextField(25);
    modelLabel = new JLabel(" Equipment Model : ");

    locationField = new JTextField(25);
    locationLabel = new JLabel(" Equipment Location: ");

    runtimeField = new JTextField(25);
    runtimeLabel = new JLabel(" Equipment Runtime (Hours) : ");

    namePanel = new JPanel();
    fieldPanel = new JPanel();

    namePanel.setLayout(new GridLayout(0, 1));
    fieldPanel.setLayout(new GridLayout(0, 1));

    namePanel.add(serialNumberLabel);
    namePanel.add(stockNumberLabel);
    namePanel.add(nameLabel);
    namePanel.add(typeLabel);
    namePanel.add(dateLabel);
    namePanel.add(manufacturerLabel);
    namePanel.add(modelLabel);
    namePanel.add(locationLabel);
    namePanel.add(runtimeLabel);

```

```

fieldPanel.add(serialNumberField);
fieldPanel.add(stockNumberField);
fieldPanel.add(nameField);
fieldPanel.add(typeField);
fieldPanel.add(dateField);
fieldPanel.add(manufacturerField);
fieldPanel.add(modelField);
fieldPanel.add(locationField);
fieldPanel.add(runtimeField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

this.setSize(800, 600);
this.getContentPane().setLayout(new BorderLayout());
this.getContentPane().add(box, BorderLayout.CENTER);
this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
this.show();

} // end EquipmentForm()

```

```

} // end class EquipmentForm

```

```
import javax.swing.*;
import java.awt.*;
```

```
/**
 * The EventForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Events Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */
```

```
public class EventForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField exerciseField;
    JLabel exerciseLabel;

    JTextField eventField;
    JLabel eventLabel;

    JComboBox typeField;
    JLabel typeLabel;

    JTextField numberField;
    JLabel numberLabel;

    JTextField durationField;
    JLabel durationLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    static String[] type = { "ANTISURFACE WARFARE",
                             "ANTISUBMARINE WARFARE",
                             "ANTIAIR WARFARE", "COMMUNICATIONS",
                             "ELECTRONIC WARFARE", "MISCELLANEOUS" };

    public EventForm(String title) {
        super(title);
    }
}
```



```

exerciseField = new JTextField(25);
exerciseLabel = new JLabel(" Exercise Name : ");

eventField = new JTextField(25);
eventLabel = new JLabel(" Event Name : ");

typeField = new JComboBox(type);
typeField.setEditable(false);
typeLabel = new JLabel(" Event Type : ");

numberField = new JTextField(25);
numberLabel = new JLabel(" Number Of Events : ");

durationField = new JTextField(25);
durationLabel = new JLabel(" Event Duration (Hours) : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(exerciseLabel);
namePanel.add(eventLabel);
namePanel.add(typeLabel);
namePanel.add(numberLabel);
namePanel.add(durationLabel);

fieldPanel.add(exerciseField);
fieldPanel.add(eventField);
fieldPanel.add(typeField);
fieldPanel.add(numberField);
fieldPanel.add(durationField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);

```

```
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end EventForm().

} // end class EventForm
```

```

import javax.swing.*;
import java.awt.*;

/**
 * The FailureForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Failures Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class FailureForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField serialField;
    JLabel serialLabel;

    JTextField failureField;
    JLabel failureLabel;

    JTextField descriptionField;
    JLabel descriptionLabel;

    JTextField diagnosisField;
    JLabel diagnosisLabel;

    JTextField dateField;
    JLabel dateLabel;

    JTextField durationField;
    JLabel durationLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    public FailureForm(String title) {

        super(title);

        serialField = new JTextField(25);
        serialLabel = new JLabel(" Equipment Serial Number : ");

        failureField = new JTextField(25);
        failureLabel = new JLabel(" Failure Number : ");
    }
}

```

```

descriptionField = new JTextField(25);
descriptionLabel = new JLabel(" Failure Description : ");

diagnosisField = new JTextField(25);
diagnosisLabel = new JLabel(" Failure Diagnosis : ");

dateField = new JTextField(25);
dateLabel = new JLabel(" Failure Date : ");

durationField = new JTextField(25);
durationLabel = new JLabel(" Failure Duration (Hours) : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(serialLabel);
namePanel.add(failureLabel);
namePanel.add(descriptionLabel);
namePanel.add(diagnosisLabel);
namePanel.add(dateLabel);
namePanel.add(durationLabel);

fieldPanel.add(serialField);
fieldPanel.add(failureField);
fieldPanel.add(descriptionField);
fieldPanel.add(diagnosisField);
fieldPanel.add(dateField);
fieldPanel.add(durationField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

```

```
        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end FailureForm()

} // end class FailureForm
```

```

import javax.swing.*;
import java.awt.*;

/**
 * The LanguageForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the ForeignLanguages Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class LanguageForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField militaryIDField;
    JLabel militaryIDLabel;

    JComboBox languageField;
    JLabel languageLabel;

    JComboBox degreeField;
    JLabel degreeLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    static String[] language = { "English", "French", "German",
        "Spanish", "Russian", "Japanese",
        "Turkish", "Chinese" };

    static String[] degree = { "A", "B", "C", "D", "F" };

    public LanguageForm(String title) {

        super(title);

        militaryIDField = new JTextField(25);
        militaryIDLabel = new JLabel(" Personnel Military ID : ");

        languageField = new JComboBox(language);
        languageField.setEditable(true);
        languageLabel = new JLabel(" Foreign Language : ");
    }
}

```

```

degreeField = new JComboBox(degree);
degreeField.setEditable(false);
degreeLabel = new JLabel(" Degree : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(militaryIDLabel);
namePanel.add(languageLabel);
namePanel.add(degreeLabel);

fieldPanel.add(militaryIDField);
fieldPanel.add(languageField);
fieldPanel.add(degreeField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

this.setSize(800, 600);
this.getContentPane().setLayout(new BorderLayout());
this.getContentPane().add(box, BorderLayout.CENTER);
this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
this.show();

} // end LanguageForm()

} // end class LanguageForm

```

```

import javax.swing.*;
import java.awt.*;

/**
 * The OperationForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Operation Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class OperationForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField nameField;
    JLabel nameLabel;

    JComboBox typeField;
    JLabel typeLabel;

    JTextField startDateField;
    JLabel startDateLabel;

    JTextField endDateField;
    JLabel endDateLabel;

    JTextField durationField;
    JLabel durationLabel;

    JTextField placeField;
    JLabel placeLabel;

    JTextField daytimeField;
    JLabel daytimeLabel;

    JTextField nighttimeField;
    JLabel nighttimeLabel;

    JTextField heloField;
    JLabel heloLabel;

    JTextField flyingField;
    JLabel flyingLabel;

    JTextField dippingNumberField;
    JLabel dippingNumberLabel;

    JTextField dippingTimeField;
    JLabel dippingTimeLabel;

```



```

JTextField fuelCostField;
JLabel fuelCostLabel;

JTextField ammoCostField;
JLabel ammoCostLabel;

JTextField amortizationField;
JLabel amortizationLabel;

JTextField costField;
JLabel costLabel;

JPanel buttonPanel;
JButton addButton;
JButton deleteButton;
JButton updateButton;
JButton cancelButton;

Color buttonColor = new Color(160, 220, 245);

static String[] type = { "INDEPENDENT", "TYPE EXERCISE",
                        "SQUADRON EXERCISE", "FLEET EXERCISE" };

public OperationForm(String title) {

    super(title);

    nameField = new JTextField(25);
    nameLabel = new JLabel(" Exercise Name : ");

    typeField = new JComboBox(type);
    typeField.setEditable(false);
    typeLabel = new JLabel(" Exercise Type : ");

    startDateField = new JTextField(25);
    startDateLabel = new JLabel(" Start Date : ");

    endDateField = new JTextField(25);
    endDateLabel = new JLabel(" End Date : ");

    durationField = new JTextField(25);
    durationLabel = new JLabel(" Duration (Days) : ");

    placeField = new JTextField(25);
    placeLabel = new JLabel(" Place (Sea/Ocean): ");

    daytimeField = new JTextField(25);
    daytimeLabel = new JLabel(" Daytime Underway Hours : ");

    nighttimeField = new JTextField(25);
    nighttimeLabel = new JLabel(" Nighttime Underway Hours : ");
}

```

```

heloField = new JTextField(25);
heloLabel = new JLabel(" Helo Tail Number : ");

flyingField = new JTextField(25);
flyingLabel = new JLabel(" Helo Flying Time (Hours) : ");

dippingNumberField = new JTextField(25);
dippingNumberLabel = new JLabel(" Number Of Dippings : ");

dippingTimeField = new JTextField(25);
dippingTimeLabel = new JLabel(" Total Dipping Time (Hours) :

fuelCostField = new JTextField(25);
fuelCostLabel = new JLabel(" Fuel Cost : ");

ammoCostField = new JTextField(25);
ammoCostLabel = new JLabel(" Ammunition Cost : ");

amortizationField = new JTextField(25);
amortizationLabel = new JLabel(" Amortization : ");

costField = new JTextField(25);
costLabel = new JLabel(" Cost Of Exercise : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(nameLabel);
namePanel.add(typeLabel);
namePanel.add(startDateLabel);
namePanel.add(endDateLabel);
namePanel.add(durationLabel);
namePanel.add(placeLabel);
namePanel.add(daytimeLabel);
namePanel.add(nighttimeLabel);
namePanel.add(heloLabel);
namePanel.add(flyingLabel);
namePanel.add(dippingNumberLabel);
namePanel.add(dippingTimeLabel);
namePanel.add(fuelCostLabel);
namePanel.add(ammoCostLabel);
namePanel.add(amortizationLabel);
namePanel.add(costLabel);

fieldPanel.add(nameField);
fieldPanel.add(typeField);
fieldPanel.add(startDateField);
fieldPanel.add(endDateField);
fieldPanel.add(durationField);
fieldPanel.add(placeField);
fieldPanel.add(daytimeField);
fieldPanel.add(nighttimeField);

```

```

        fieldPanel.add(heloField);
        fieldPanel.add(flyingField);
        fieldPanel.add(dippingNumberField);
        fieldPanel.add(dippingTimeField);
        fieldPanel.add(fuelCostField);
        fieldPanel.add(ammoCostField);
        fieldPanel.add(amortizationField);
        fieldPanel.add(costField);

        box = new Box(BoxLayout.X_AXIS);
        box.add(namePanel);
        box.add(fieldPanel);

        addButton = new JButton("ADD RECORD");
        addButton.setBackground(buttonColor);

        deleteButton = new JButton("DELETE RECORD");
        deleteButton.setBackground(buttonColor);

        updateButton = new JButton("UPDATE RECORD");
        updateButton.setBackground(buttonColor);

        cancelButton = new JButton("CANCEL");
        cancelButton.setBackground(buttonColor);

        buttonPanel = new JPanel();
        buttonPanel.add(addButton);
        buttonPanel.add(deleteButton);
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end OperationForm()

} // end class OperationForm

```

```

import javax.swing.*;
import java.awt.*;

/**
 * The OverhaulForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Overhaul Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class OverhaulForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JComboBox shipField;
    JLabel shipLabel;

    JTextField numberField;
    JLabel numberLabel;

    JTextField startDateField;
    JLabel startDateLabel;

    JTextField endDateField;
    JLabel endDateLabel;

    JTextField durationField;
    JLabel durationLabel;

    JTextField shipyardField;
    JLabel shipyardLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    static String[] ship = { "TBUP" };

    public OverhaulForm(String title) {

        super(title);

        shipField = new JComboBox(ship);
        shipField.setEditable(false);
        shipLabel = new JLabel(" Ship's Int'l Callsign : ");
    }
}

```

```

numberField = new JTextField(25);
numberLabel = new JLabel(" Overhaul Number : ");

startDateField = new JTextField(25);
startDateLabel = new JLabel(" Overhaul Start Date : ");

endDateField = new JTextField(25);
endDateLabel = new JLabel(" Overhaul End Date : ");

durationField = new JTextField(25);
durationLabel = new JLabel(" Overhaul Duration (Days) : ");

shipyardField = new JTextField(25);
shipyardLabel = new JLabel(" Shipyard Name : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(shipLabel);
namePanel.add(numberLabel);
namePanel.add(startDateLabel);
namePanel.add(endDateLabel);
namePanel.add(durationLabel);
namePanel.add(shipyardLabel);

fieldPanel.add(shipField);
fieldPanel.add(numberField);
fieldPanel.add(startDateField);
fieldPanel.add(endDateField);
fieldPanel.add(durationField);
fieldPanel.add(shipyardField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);

```

```
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();
    } // end OverhaulForm()

} // end class OverhaulForm
```

```

import javax.swing.*;
import java.awt.*;

/**
 * The PersonnelForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Personnel Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */

public class PersonnelForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField militaryIDField;
    JLabel militaryIDLabel;

    JTextField firstNameField;
    JLabel firstNameLabel;

    JTextField lastNameField;
    JLabel lastNameLabel;

    JComboBox departmentField;
    JLabel departmentLabel;

    JComboBox divisionField;
    JLabel divisionLabel;

    JComboBox rankField;
    JLabel rankLabel;

    JComboBox ratingField;
    JLabel ratingLabel;

    JTextField birthDateField;
    JLabel birthDateLabel;

    JTextField birthPlaceField;
    JLabel birthPlaceLabel;

    JTextField fatherField;
    JLabel fatherLabel;

    JTextField motherField;
    JLabel motherLabel;

    JTextField serviceDateField;
    JLabel serviceDateLabel;

    JTextField rankDateField;

```

```
JLabel rankDateLabel;

JComboBox genderField;
JLabel genderLabel;

JComboBox maritalField;
JLabel maritalLabel;

JTextField spouseField;
JLabel spouseLabel;

JTextField childrenField;
JLabel childrenLabel;

JTextField streetField;
JLabel streetLabel;

JTextField cityField;
JLabel cityLabel;

JTextField stateField;
JLabel stateLabel;

JTextField zipField;
JLabel zipLabel;

JTextField phoneField;
JLabel phoneLabel;

JTextField specialityField;
JLabel specialityLabel;

JComboBox educationField;
JLabel educationLabel;

JTextField assignmentField;
JLabel assignmentLabel;

JTextField startDateField;
JLabel startDateLabel;

JTextField cabinNumberField;
JLabel cabinNumberLabel;

JTextField cabinPhoneField;
JLabel cabinPhoneLabel;

JPanel buttonPanel;
JButton addButton;
JButton deleteButton;
JButton updateButton;
JButton cancelButton;
```



```

Color buttonColor = new Color(160, 220, 245);

static String[] department = { "Operations", "Engineering",
                                "Weapons", "Electronics",
                                "Navigation", "Supply" };

static String[] division = { "CIC", "Communications",
                              "Electronic Warfare",
                              "Main Propulsion", "Electrical",
                              "Damage Control",
                              "Anti Surface Warfare",
                              "Anti Submarine Warfare",
                              "Anti Air Warfare", "Fire Control",
                              "Weapons Electronics",
                              "CIC Electronics",
                              "Communications Electronics",
                              "Administration", "Navigation",
                              "Deck", "Supply", "Medical" };

static String[] rank = { "ENSIGN", "LTJG", "LIEUTENANT",
                          "LTCDR", "COMMANDER", "CAPTAIN",
                          "PETTY OFFICER 2ND CLASS",
                          "PETTY OFFICER 1ST CLASS",
                          "CHIEF PETTY OFFICER 2ND CLASS",
                          "CHIEF PETTY OFFICER",
                          "SENIOR CHIEF PETTY OFFICER",
                          "MASTER CHIEF PETTY OFFICER" };

static String[] rating = { "OFFICER", "PETTY OFFICER",
                            "ENLISTED" };

static String[] gender = { "MALE", "FEMALE" };

```

```

static String[] marital = { "SINGLE", "MARRIED" };
static String[] education = { "HIGH SCHOOL", "COLLEGE",
                               "UNIVERSITY", "MASTER" };

public PersonnelForm(String title) {
    super(title);

    militaryIDField = new JTextField(25);
    militaryIDLabel = new JLabel(" Military ID : ");

    firstNameField = new JTextField(25);
    firstNameLabel = new JLabel(" First Name : ");

    lastNameField = new JTextField(25);
    lastNameLabel = new JLabel(" Last Name : ");

    departmentField = new JComboBox(department);
    departmentField.setEditable(false);
    departmentLabel = new JLabel(" Department : ");

    divisionField = new JComboBox(division);
    divisionField.setEditable(false);
    divisionLabel = new JLabel(" Division : ");

    rankField = new JComboBox(rank);
    rankField.setEditable(false);
    rankLabel = new JLabel(" Rank : ");

    ratingField = new JComboBox(rating);
    ratingField.setEditable(false);
    ratingLabel = new JLabel(" Rating : ");

    birthDateField = new JTextField(25);
    birthDateLabel = new JLabel(" Date Of Birth : ");

    birthPlaceField = new JTextField(25);
    birthPlaceLabel = new JLabel(" Place Of Birth : ");

    fatherField = new JTextField(25);
    fatherLabel = new JLabel(" Father's Name : ");

    motherField = new JTextField(25);
    motherLabel = new JLabel(" Mother's Name : ");

    serviceDateField = new JTextField(25);
    serviceDateLabel = new JLabel(" Active Duty Service Date :

    rankDateField = new JTextField(25);
    rankDateLabel = new JLabel(" Date Of Rank : ");

```

```

genderField = new JComboBox(gender);
genderField.setEditable(false);
genderLabel = new JLabel(" Gender : ");

maritalField = new JComboBox(marital);
maritalField.setEditable(false);
maritalLabel = new JLabel(" Marital Status : ");

spouseField = new JTextField(25);
spouseLabel = new JLabel(" Spouse's Name : ");

childrenField = new JTextField(25);
childrenLabel = new JLabel(" Number Of Children : ");

streetField = new JTextField(25);
streetLabel = new JLabel(" Street : ");

cityField = new JTextField(25);
cityLabel = new JLabel(" City : ");

stateField = new JTextField(25);
stateLabel = new JLabel(" State : ");

zipField = new JTextField(25);
zipLabel = new JLabel(" Zip Code : ");

phoneField = new JTextField(25);
phoneLabel = new JLabel(" Phone Number : ");

specialityField = new JTextField(25);
specialityLabel = new JLabel(" Speciality : ");

educationField = new JComboBox(education);
educationField.setEditable(false);
educationLabel = new JLabel(" Education : ");

assignmentField = new JTextField(25);
assignmentLabel = new JLabel(" Current Assignment : ");

startDateField = new JTextField(25);
startDateLabel = new JLabel(" Start Date : ");

cabinNumberField = new JTextField(25);
cabinNumberLabel = new JLabel(" Cabin Number : ");

cabinPhoneField = new JTextField(25);
cabinPhoneLabel = new JLabel(" Cabin Phone : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(militaryIDLabel);

```

```
namePanel.add(firstNameLabel);
namePanel.add(lastNameLabel);
namePanel.add(departmentLabel);
namePanel.add(divisionLabel);
namePanel.add(rankLabel);
namePanel.add(ratingLabel);
namePanel.add(birthDateLabel);
namePanel.add(birthPlaceLabel);
namePanel.add(fatherLabel);
namePanel.add(motherLabel);
namePanel.add(serviceDateLabel);
namePanel.add(rankDateLabel);
namePanel.add(genderLabel);
namePanel.add(maritalLabel);
namePanel.add(spouseLabel);
namePanel.add(childrenLabel);
namePanel.add(streetLabel);
namePanel.add(cityLabel);
namePanel.add(stateLabel);
namePanel.add(zipLabel);
namePanel.add(phoneLabel);
namePanel.add(specialityLabel);
namePanel.add(educationLabel);
namePanel.add(assignmentLabel);
namePanel.add(startDateLabel);
namePanel.add(cabinNumberLabel);
namePanel.add(cabinPhoneLabel);
```

```
fieldPanel.add(militaryIDField);
fieldPanel.add(firstNameField);
fieldPanel.add(lastNameField);
fieldPanel.add(departmentField);
fieldPanel.add(divisionField);
fieldPanel.add(rankField);
fieldPanel.add(ratingField);
fieldPanel.add(birthDateField);
fieldPanel.add(birthPlaceField);
fieldPanel.add(fatherField);
fieldPanel.add(motherField);
fieldPanel.add(serviceDateField);
fieldPanel.add(rankDateField);
fieldPanel.add(genderField);
fieldPanel.add(maritalField);
fieldPanel.add(spouseField);
fieldPanel.add(childrenField);
fieldPanel.add(streetField);
fieldPanel.add(cityField);
fieldPanel.add(stateField);
fieldPanel.add(zipField);
fieldPanel.add(phoneField);
fieldPanel.add(specialityField);
fieldPanel.add(educationField);
fieldPanel.add(assignmentField);
fieldPanel.add(startDateField);
fieldPanel.add(cabinNumberField);
```

```

        fieldPanel.add(cabinPhoneField);

        box = new Box(BoxLayout.X_AXIS);
        box.add(namePanel);
        box.add(fieldPanel);

        addButton = new JButton("ADD RECORD");
        addButton.setBackground(buttonColor);

        deleteButton = new JButton("DELETE RECORD");
        deleteButton.setBackground(buttonColor);

        updateButton = new JButton("UPDATE RECORD");
        updateButton.setBackground(buttonColor);

        cancelButton = new JButton("CANCEL");
        cancelButton.setBackground(buttonColor);

        buttonPanel = new JPanel();
        buttonPanel.add(addButton);
        buttonPanel.add(deleteButton);
        buttonPanel.add(updateButton);
        buttonPanel.add(cancelButton);

        this.setSize(800, 600);
        this.getContentPane().setLayout(new BorderLayout());
        this.getContentPane().add(box, BorderLayout.CENTER);
        this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        this.show();

    } // end PersonnelForm()

} // end class PersonnelForm

```

```
import javax.swing.*;
import java.awt.*;
```

```
/**
```

```
 * The PortVisitForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the PortVisits Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */
```

```
public class PortVisitForm extends JFrame {
```

```
    Box box;
    JPanel namePanel;
    JPanel fieldPanel;
```

```
    JTextField exerciseField;
    JLabel exerciseLabel;
```

```
    JTextField portField;
    JLabel portLabel;
```

```
    JTextField startDateField;
    JLabel startDateLabel;
```

```
    JTextField endDateField;
    JLabel endDateLabel;
```

```
    JTextField durationField;
    JLabel durationLabel;
```

```
    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;
```

```
    Color buttonColor = new Color(160, 220, 245);
```

```
    public PortVisitForm(String title) {
```

```
        super(title);
```

```
        exerciseField = new JTextField(25);
        exerciseLabel = new JLabel(" Exercise Name : ");
```

```
        portField = new JTextField(25);
        portLabel = new JLabel(" Port Name : ");
```

```
        startDateField = new JTextField(25);
        startDateLabel = new JLabel(" Visit Start Date : ");
```

```
        endDateField = new JTextField(25);
```

```

endDateLabel = new JLabel(" Visit End Date : ");

durationField = new JTextField(25);
durationLabel = new JLabel(" Visit Duration (Days) : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(exerciseLabel);
namePanel.add(portLabel);
namePanel.add(startDateLabel);
namePanel.add(endDateLabel);
namePanel.add(durationLabel);

fieldPanel.add(exerciseField);
fieldPanel.add(portField);
fieldPanel.add(startDateField);
fieldPanel.add(endDateField);
fieldPanel.add(durationField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

this.setSize(800, 600);
this.getContentPane().setLayout(new BorderLayout());
this.getContentPane().add(box, BorderLayout.CENTER);
this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
this.show();

} // end PortVisitForm()

} // end class PortVisitForm

```

```

import javax.swing.*;
import java.awt.*;

/**
 * The TrainingForm class inherits from the JFrame class and provides
 * a graphical representation, which consists of labels, fields, and
 * combo boxes, for the Training Table in the POET Database.
 *
 * @author LTJG. Yuksel Can
 */
public class TrainingForm extends JFrame {

    Box box;
    JPanel namePanel;
    JPanel fieldPanel;

    JTextField nameField;
    JLabel nameLabel;

    JComboBox placeField;
    JLabel placeLabel;

    JTextField durationField;
    JLabel durationLabel;

    JTextField descriptionField;
    JLabel descriptionLabel;

    JPanel buttonPanel;
    JButton addButton;
    JButton deleteButton;
    JButton updateButton;
    JButton cancelButton;

    Color buttonColor = new Color(160, 220, 245);

    static String[] place = { "YILDIZLAR TRAINING CENTER",
                              "KARAMURSEL TRAINING CENTER",
                              "DERINCE TRAINING CENTER" };

    public TrainingForm(String title) {

        super(title);

        nameField = new JTextField(25);
        nameLabel = new JLabel(" Course Name : ");

        placeField = new JComboBox(place);
        placeField.setEditable(false);
        placeLabel = new JLabel(" Training Center : ");
    }
}

```



```

durationField = new JTextField(25);
durationLabel = new JLabel(" Course Duration (Weeks) : ");

descriptionField = new JTextField(25);
descriptionLabel = new JLabel(" Course Description : ");

namePanel = new JPanel();
fieldPanel = new JPanel();

namePanel.setLayout(new GridLayout(0, 1));
fieldPanel.setLayout(new GridLayout(0, 1));

namePanel.add(nameLabel);
namePanel.add(placeLabel);
namePanel.add(durationLabel);
namePanel.add(descriptionLabel);

fieldPanel.add(nameField);
fieldPanel.add(placeField);
fieldPanel.add(durationField);
fieldPanel.add(descriptionField);

box = new Box(BoxLayout.X_AXIS);
box.add(namePanel);
box.add(fieldPanel);

addButton = new JButton("ADD RECORD");
addButton.setBackground(buttonColor);

deleteButton = new JButton("DELETE RECORD");
deleteButton.setBackground(buttonColor);

updateButton = new JButton("UPDATE RECORD");
updateButton.setBackground(buttonColor);

cancelButton = new JButton("CANCEL");
cancelButton.setBackground(buttonColor);

buttonPanel = new JPanel();
buttonPanel.add(addButton);
buttonPanel.add(deleteButton);
buttonPanel.add(updateButton);
buttonPanel.add(cancelButton);

this.setSize(800, 600);
this.getContentPane().setLayout(new BorderLayout());
this.getContentPane().add(box, BorderLayout.CENTER);
this.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
this.show();

} // end TrainingForm()

} // end class TrainingForm

```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Elmasri, R., and Navathe, S. B., *Fundamentals of Database Systems*, 2nd Edition, Addison-Wesley Publishing Company, Inc., 1994.
2. Date, C. J., *An Introduction to Database Systems*, 5th Edition, Addison-Wesley Publishing Company, Inc., 1990.
3. Kroenke, D. M., *Database Processing, Fundamentals, Design, and Implementation*, 6th Edition, Prentice Hall, 1998.
4. Akin, R., and O'Brian, F.P., *Analysis of Java Distributed Architectures in Designing and Implementing a Client/Server Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1998.
5. Akbay, M., and Lewis, S.C., *Design and Implementation of an Enterprise Information System Utilizing a Component Based Three-Tier Client/Server Database System*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1999.
6. Teorey, T. J., *Database Modeling and Design: The Fundamental Principles*, 2nd Edition, Morgan Kaufmann Publishers, Inc., 1994.
7. Cassel, P., *Teach Yourself Access 97 in 14 Days*, 4th Edition, Sams Publishing, 1996.
8. Prague, C. N., and Irwin, M. R., *Access 97 Bible*, IDG Books Worldwide, Inc., 1997.
9. Hammer, M., and McLeod, D., *Database Description with SDM: A Semantic Database Model*, ACM Transactions on Database Systems, Vol. 6, No. 3, September 1981.
10. Hamilton, G., Cattel, R., Fisher, M., *JDBC Database Access with Java*, Addison-Wesley Publishing Company, Inc., 1997.
11. Horstmann, C. S., and Cornell, G., *Core Java 2*, Sun Microsystems, 1999.
12. JavaSoft, *JDBC: A Java SQL API*, Sun Microsystems, January 1997.
13. Senn, James A., *Analysis and Design of Information Systems*, McGraw-Hill Book Company, 1984.

14. Daft, R. L., *Organization Theory and Design*, 6th Edition, South-Western College Publishing, 1998.
15. Whitten, J.L., Bentley, L.D., Barlow, V.M., *Systems Analysis and Design Methods*, 3rd Edition, Irwin, 1994.
16. Danziger, J. N., and Kraemer, K. L., *People and Computers*, Columbia University Press, 1986.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center2
8725 John J. Kingman Road, Ste 0944
Ft. Belvoir, VA 22060-6218
2. Dudley Knox Library2
Naval Postgraduate School
411 Dyer Rd.
Monterey, CA 93943-5101
3. Deniz Kuvvetleri Komutanligi1
Personel Daire Baskanligi
Bakanliklar
Ankara, TURKEY
4. Deniz Kuvvetleri Komutanligi1
Kutuphanesi
Bakanliklar
Ankara, TURKEY
5. Deniz Harp Okulu2
Kutuphanesi
Tuzla
Istanbul, TURKEY
6. Chairman, Code CS1
Naval Postgraduate School
Monterey, CA 93943-5101

7. Chairman, Code SM.....1
Naval Postgraduate School
Monterey, CA 93943-5101
8. Prof. C. Thomas Wu (CS/Wu).....1
Naval Postgraduate School
Monterey, CA 93943-5100
9. Prof. Lee Edwards (SM/Ed).....1
Naval Postgraduate School
Monterey, CA 93943-5100
10. Yazilim Gelistirme Grup Baskanligi1
Deniz Harp Okulu Komutanligi
Tuzla
Istanbul, TURKEY
11. LTJG. Yuksel Can2
Findikli Mahallesi
Limon Sokak No: 14 Daire 3
Maltepe
Istanbul, TURKEY