THE RE-ENGINEERING OF THE AIR FORCE INSTITUTEOF TECHNOLOGY STUDENT INFORMATION SYSTEM

THESIS

Douglas James Wu, Captain, USAF

AFIT/GCS/ENG/94D-27

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-27

THE RE-ENGINEERING OF THE AIR FORCE INSTITUTEOF TECHNOLOGY STUDENT INFORMATION SYSTEM

THESIS

Douglas James Wu, Captain, USAF

AFIT/GCS/ENG/94D-27

Approved for public release; distribution unlimited

AFIT/GCS/ENG/94D-27

THE RE-ENGINEERING OF THE AIR FORCE INSTITUTE OF TECHNOLOGY STUDENT INFORMATION SYSTEM

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Masters of Science in Computer Science

Douglas J. Wu, B.S.

Captain, USAF

DECEMBER, 1994

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

ACKNOWLEDGMENTS

I would like to thank my advisor, Lieutenant Colonel Patricia Lawlis, for her guidance and assistance during this research effort. I also wish to thank my readers, Lieutenant Colonel Mark Roth and Doctor Eugene Santos.

Additional thanks to my soul mate and ray of sunshine, Priscilla, for her support and understanding.

Douglas J. Wu

Table of Contents

Page

Acknowledgements iii

List of Figures vi

List of Tables ix

Abstract x

I. Introduction 1-1
- Background 1-1
- Problem 1-2
- Hypothesis 1-2
- Research Objectives 1-2
- Assumptions 1-2
- Sequence of Presentation 1-3

II. Summary of Current Knowledge 2-1
- Topic Statement and Key Terms 2-1
- Treatment and Organization 2-1
- Discussion of the Literature 2-2
  - Object-Oriented Methodology 2-2
  - Relational Databases 2-3
  - Object-Oriented Databases 2-5
  - OODBMS versus RDBMS 2-6
- Conclusion 2-9

III. Analysis and Design 3-1
- Introduction 3-1
- Analysis 3-2
  - Problem Statement 3-3
  - Object Model 3-4
    - Identify Object Classes 3-4
    - Identify Associations 3-6
    - Identify Attributes 3-7
    - Access Paths for Likely Queries 3-10
    - Refine Object Model 3-11
  - Dynamic Model 3-11
    - Scenarios 3-11

Events Between Objects 3-13
Event Flow Diagrams 3-14
State Diagrams 3-15
Verify Consistency 3-19
Functional Model 3-19
Input and Output Variables 3-20
Data Flow Diagrams (DFD) 3-20
Constraints 3-22
Optimization Criteria 3-22
Final Analysis 3-23
Add Functional Model Operations 3-23
Compare Models with Problem Statement 3-23
Develop More Detailed Scenarios 3-24
System Design 3-24
Object Model Mapped to Database 3-25
Mapping Object Classes 3-27
Mapping Ternary Associations 3-29
Mapping associations with link attributes 3-36
Summary 3-37

IV. Implementation Issues 4-1
Introduction 4-1
Student Class Schedule 4-1
SQL versus OQL Comparison 4-3
Class Roster 4-8
SQL versus OQL 4-10
Summary 4-11

V. Summary and Conclusions 5-1
Summary 5-1
Conclusions 5-3
Future Plans for AFITSIS 5-4

Appendix A - AFITSIS Design A-1

Appendix B - Relational Tables B-1

Appendix C - List of Abbreviations C-1

Bibliography

Vita

List of Figures

Figure Page

3.1 Overview of the analysis process 3-1

3.2 Student Tracking and Registration System Problem Statement 3-3

3.3 Candidate object classes 3-4

3.4 Current Object Classes 3-6

3.5 Candidate Associations 3-7

3.6 Relational table for Person 3-8

3.7 One-to-many Association 3-8

3.8 Ternary Association 3-8

3.9 Inheritance 3-9

3.10 Multiple inheritance 3-9

3.11 Association with link attribute 3-10

3.12 Aggregation 3-10

3.13 Faculty User Scenario 3-12

3.14 Administrative User Scenario 3-12

3.15 Event trace for faculty user scenario 3-13

3.16 Event trace for administrative user scenario 3-14

3.17 Event flow diagram for scenarios 3-14

3.18 Application state diagram 3-15

3.19 Substates of In_Use state 3-16

3.20 Present Options Menu state as dependent state 3-17

3.21 Partial substates for Process Selection state 3-17

3.22 Input and output values for application 3-20

3.23 STARS Application Level 0 DFD 3-21
3.24 Perform Action Level 1 DFD 3-21
3.25 Validate User Login Level 2 DFD 3-21
3.26 Function description for Validate User Login process 3-22
3.27 Comparison of Database Management System Architectures 3-26
3.28 Mapping object class to relational database 3-28
3.29 Mapping object class to object-oriented database 3-29
3.30 Mapping ternary association to relational database 3-30
3.31 Ternary association with binary relationships 3-31
3.32 ODL for ternary association modeled with binary relationships 3-32
3.33 Abstracting the ternary relationship as an object 3-33
3.34 ODL for abstract ternary relationship 3-33
3.35 Mapping ternary association to object-oriented database 3-35
3.36 Mapping association with link attribute to object database 3-37
4.1 Student class schedule 4-2
4.2 Student class schedule object model 4-2
4.3 Class roster 4-9
4.4 Class Roster Object Model 4-9
A.1 School Object Diagram A-2
A.2 Person Object Diagram A-3
A.3 Multiple Inheritance Person Object Diagram A-4
A.4 Course Object Diagram A-5
A.5 Student user scenario A-5
A.6 Event trace for student user scenario A-6
A.7 Application event flow diagram A-6

A.8 Application state diagram A-7
A.9 Substates of In_Use state A-7
A.10 Possible generalization of Present Options Menu State A-7
A.11 Partial substates of Process Selection state A-8
A.12 Process Selection state A-8
A.13 State diagram for Database System A-10
A.14 STARS Application Level 0 DFD A-10
A.15 Perform Action Level 1 DFD A-10
A.16 Offer Menu Options Level 2 DFD A-11
A.17 Handle Selection Level 2 DFD A-11
A.18 Do Event Level 3 DFD A-12
A.19 Create Level 4 DFD A-12
A.20 Modify Level 4 DFD A-13
A.21 Save Level 5 DFD A-13
A.22 Display Level 4 DFD A-13
A.23 Print Level 4 DFD A-14
A.24 Delete Level 4 DFD A-14
A.25 Mapping association to relational database A-15
A.26 Mapping association to object-oriented database A-16
A.27 Student Class Schedule Object Model A-17
A.28 Class Roster Object Model A-24
A.29 Assign Student Academic Advisor Object Model A-30
A.30 AFIT Course Catalog Object Model A-34
A.31 Student Transcript Object Model A-35

List of Tables

Table Page

3.1 Partial state transition table for Application 3-19

A.1 State transition table for Application A-9

A.2 State transition table for Database System A-10

Abstract

This research describes the design and implementation issues associated with re-engineering the Air Force Institute of Technology Student Information System (AFITSIS). Currently, AFITSIS executes on aging relational database technology and has unfriendly user interface mechanisms. The two research objectives met were to research current AFITSIS requirements, design, and implementation, and use object-orient methods to design an alternative implementation based on proposed object database management system standards. This research explores how AFITSIS performance and capabilities might be enhanced by taking advantage of new object-oriented software engineering techniques. One of the primary benefits of this research is a detailed object modeling technique analysis and design that may be used as a foundation for upgrading the current AFITSIS.

THE RE-ENGINEERING OF THE AIR FORCE INSTITUTE OF TECHNOLOGY STUDENT INFORMATION SYSTEM

I. Introduction

Many information systems suffer from old technology which no longer adequately meets the needs of the users. The Air Force Institute of Technology Student Information System (AFITSIS) is such a system. This research explored how AFITSIS's performance and capabilities might be enhanced by taking advantage of new object-oriented software engineering techniques.

Background. In 1987 the Air Force Institute of Technology (AFIT) contracted Systems Research Laboratories, Inc. to develop an automated system, called the Student Tracking and Registration System (STARS), for scheduling courses, registering students for courses, tracking students' academic histories, and generating related reports. The STARS application uses the Structured Query Language (SQL) to access an Oracle Relational Database Management System (RDBMS) [21:1].

In the seven years since the initial STARS development three other applications have been developed (and others continue to be developed) sharing tables with relevant attributes within the RDBMS [11]. This group of applications is referred to as AFITSIS and is accessible to any user with a designated Username and Password through the AFITNET communications network [6]. In addition to the AFITSIS applications there are a number of other applications that share student-related information from the Oracle RDBMS. The 8-member AFIT/SCV organization is responsible for providing maintenance and upgrades to all applications as requested by users from all administrative offices as well as from faculty and staff of AFIT's schools [11]. Planned expansion of AFIT's educational boundaries for the Dayton Area Graduate Studies Institute (DAGSI), to include Wright State University and the University of Dayton graduate students, will no doubt increase the requirements of the current DBMS.

Problem. AFITSIS is currently designed and implemented upon aging relational database technology and unfriendly user interface mechanisms.

Hypothesis. AFITSIS capabilities and performance can be substantially improved by: (1) redesigning the system using an object-oriented design methodology, (2) converting from the relational database management system to a new object-oriented database management system, and (3) implementing a user interface in a Windows environment. These steps will not only provide a basis for an improved system now, it will also aid software maintainers faced with adding capabilities in the future.

Research Objectives. In order to address the problem stated above and establish the validity of the above hypothesis, the following research objectives were established:

1. Research current AFITSIS requirements, design, and implementation. Document current system performance benchmarks and maintainability metrics for comparison with a prototype system.

2. Use object-oriented methods to design and implement an object-oriented database management system (OODBMS) prototype system. Use current requirements as stated by AFIT/SC and STARS users (performance and user interface requirements). Create the design structure with possible upgrades in mind.

3. Test OODBMS prototype against current requirements. Compare performance benchmarks and maintainability predictions with current RDBMS.

In the end, a working prototype proved impractical to develop as a part of this thesis. However, this thesis development produced detailed design information adequate for a straightforward prototype development.

Assumptions. The following assumptions were made during the thesis effort:

1. Access to AFITSIS database and query information is available.

2. Access to an OODBMS is available for use with the prototype to be developed.

3. The OODBMS will work with a Windows-based prototype.

Assumption two proved to be only partly true. Ongoing commercial development should soon result in an ODBMS adequate for prototyping. However, the currently available ODBMSs still do not support the interface standard required to support implementation of a useful prototype.

Sequence of Presentation. The thesis is divided into five chapters. Chapter I, *Introduction*, has provided an overview of the work. Chapter II, *Summary of Current Knowledge*, discusses the background information which provides the foundation for this thesis research. Chapter III, *Analysis and Design*, expands in detail the analysis performed by designing an Object Modeling Technique (OMT) model for this research. Chapter IV, *Implementation Issues*, discusses important issues relating to the implementation of a subset of the object classes contained in STARS. Lastly, Chapter V, *Summary and Conclusions*, summarizes the results of the research, draws conclusions from the summary, and makes recommendations for future research in this area.

II. Summary of Current Knowledge

Topic Statement and Key Terms. This literature review provides the foundation on which to begin the re-engineering and improvement of AFITSIS. AFITSIS is a software database application used by AFIT (orderly room, faculty, administration, etc.) to query, archive, and output data on past and present students, faculty, administration personnel, curriculum, schedules, and assignments [21:1]. The four applications forming AFITSIS are STARS, QUEST, CCQ, and ISA. Their functions are described briefly below [6]:

STARS maintains AFIT student information from admittance to graduation.

QUEST, QUota Education and Selection Transactions, tracks information related to students' selection for AFIT or Civilian Institution program, and associated Air University quotas.

CCQ, maintains orderly room applicable information, such as box numbers, locker numbers, sections leaders, security access badges, building access cards, weigh-in, aerobic data, weight management program statistics, emergency locator information, and other current student information.

ISA, International Student Affairs, maintains information on international students attending AFIT, such as family, funding, sponsor, and program data.

Treatment and Organization. The Discussion of the Literature section is divided into four subsections labeled Object-Oriented Methodology, Relational Databases, Object-Oriented Databases, and OODBMS versus RDBMS. The Object-Oriented Methodology subsection describes the stages used by the developer to analyze a problem, design a system, and implement the system into a usable product. The Relational Database subsection is required to understand how current relational systems operate, so AFITSIS capabilities may be provided for in an object-oriented system. The Object-

Oriented Database subsection discusses engineering features important to developers and the system functions required by the end users. Lastly, the OODBMS versus RDBMS subsection outlines the facilities that will be required for the next generation of database technology.

Discussion of the Literature.

Object-Oriented Methodology. Object-oriented methodologies allow developers to analyze problems and divide them into entities residing in specific states and exhibiting certain dynamic behaviors. The entities become objects in the system. The designer further defines the relationships between the objects to determine how the system functions as a whole [12:1-2]. The object-oriented methodology used in this research is OMT described in *Object-Oriented Modeling and Design* [19:4-6]. The four specific OMT stages are:

1. Analysis. During the analysis stage, the developer defines the system requirements. Objects are identified and their relationships to other objects are recorded. Implementation decisions are specifically avoided. OMT recommends defining three models during analysis: an object relationship model, a dynamic relationship model, and a functional model.

2. System Design. In this stage the system's architecture is determined. The application is broken into subsystems, and the subsystems into sub-subsystems (if required). Resources (such as storage, processors, etc.) are allocated to each subsystem. Control mechanisms (procedural, event-driven, etc.) are defined for each subsystem. The overall focus is on what needs to be done, independently of how it is to be done.

3. Object Design. During the object design phase, the object relationship model, dynamic model, and functional model are evaluated to determine what operations must be

implemented for each object. Algorithms for these operations are designed. Structures for representing the relationships between objects are defined.

4. Implementation. The final stage of OMT involves transforming the design into an executable system. This is dependent on whether the software language selected supports object-oriented programming. Chapter IV discusses implementation of object models from the student tracking and registration system into the relational database Structured Query Language and into the proposed object database Object Query Language (OQL).

Relational Databases. Traditional database applications are built around relational database management systems where data is stored in two-dimensional tables[2:45]. It is a cumbersome approach in which data has two distinct representations between in-memory or stored on-disk. The DBMS queries and shared access support only record-based disk stored data. To read data from the database, the programmer needs to allocate a buffer and issue a DBMS query. The DBMS selects the record and copies it to the buffer. The programmer treats the data stored in the buffer as though it is an instance of the programming language type by mapping the data structure declaration for the type (within the program) over the buffer. Type safety is lost in the interface and problems that may arise include, buffers that are too small or too large, misaligning the overlaid template, or defined template fields not agreeing with the record representation chosen by the database [3:32-33].

Queries returning sets of records require a cursor mechanism to coordinate between the programming language runtime and the DBMS runtime as the programmer iterates through the resultant set; mapping one record at a time into his buffer. Programs using anything more complex than records requires the programmer to write code reassembling the database records into the program's data structure [3:33]. All of this

requires time and makes relational databases less desirable for many real world applications [2:45].

Relational database management systems are derived from strict mathematical concepts [18:162]. These systems evolved to support such common database features as [16:425]:

- Uniformity. Large numbers of similarly structured data items.
- Fixed record size. Data items with fixed record lengths; approximately the same number of bytes.
- Small data items. Records rarely larger than 300 bytes long.
- Atomic fields. Short fixed-length record fields.
- Short transactions. Generally little or no human interaction; users prepare transactions, submit for execution, and await the outcome.
- Static schemes. Database schemes changed infrequently and changes are simple, such as create a relation, remove a relation, add attributes, or remove attributes.

These features make relational databases less capable of dealing with data types not easily manipulated with conventional functions, such as those used in manufacturing, engineering, software management, and document management [18:162]. The new systems require the ability to handle evolving features, such as [16:426-427]:

- Complex objects. Objects modeling the real-world with relationships stored within objects and nested relationships stored within relationships.
- Behavioral data. Objects responding to the same command in different ways.
- Meta knowledge. General rules of the application which maybe represented with the database.
- Long-duration transactions. Those seen more often with Computer-Aided Design (CAD) and Computer-Aided Software Engineering (CASE) applications where

human interaction often leads to modification of design. These cause transaction aborts, waits for locks, and are more complex than short duration business-type transactions.

Object-Oriented Databases. Object-oriented techniques give a developer powerful tools to define requirements and translate them into working code. These techniques are valuable not only in programming languages, but also in defining the next generation of database systems -- object-oriented databases [4:33-34]. It is open to discussion as to what precise features a database must contain to be considered an object-oriented database. One author maintains an object-oriented database management system must support complex objects, unique object identifiers, object encapsulation, data types, inheritance, persistence, extendibility, and other software engineering features [1:27]. All of these software engineering features are invisible to the database management system user, but are important in supporting the database management systems' main responsibilities: sharing files among users, ensuring data integrity and recovery with failures, distributing data in a network, and managing the search through large amounts of data [2:44-45].

Two years ago in an effort to establish an industry-wide agreement for object-oriented database technology a consortium of object-oriented database companies formed the Object Database Management Group (ODMG). This group of technical experts published *The Object Database Standard: ODMG-93* which defines a standard for object database management systems [5:1-15]. This standard is referred to throughout the remainder of the text as the ODMG-93 standard or simply ODMG-93. The primary goal of ODMG-93 was to introduce proposed standards allowing portable applications to be written. For this goal to be achieved the data schema, programming language binding, data manipulation, and data query languages must be portable. This goal of source code

portability makes database objects appear as programming language objects, in one or more existing programming languages [5:2-3].

OODBMS versus RDBMS. For the past three decades applications have grown in functionality, and the cost of implementing, maintaining, and extending them has risen while software has evolved through four generations of database technology: file, hierarchical, CODASYL, and relational. The evolution of computer applications and database technology requires a management system that can meet both today's and newly emerging requirements. Since the late 1970s relational products have handled most conventional information systems applications, but appear not as adept for new and future information system needs. These applications, such as computer-aided design, engineering, and manufacturing software systems, require the ability to create, access, manipulate, and save large amounts of persistent data as well as be available to a large and diverse number of users. The next generation of database technology must build upon the conventional database technologies meeting today's requirements and exploit current technology to meet the growing new requirements. The database technology must provide the capabilities required by conventional information systems before they can successfully evolve to the fifth generation database technology [14:35-36].

The two contending database technologies to capture this next generation appear to be extended relational database technology and object-oriented database technology. The extended relational approach extends the current relational model of data and provides a query language to broaden the applicability of the model without sacrificing the relational foundation. Extended relational data models allow recursive queries, relations that are not in first normal form, complex objects, and the combining of logic-base rules with triggers. The object-oriented approach starts with an object-oriented data model and a language that captures it and extends them so that database objects appear as

programming language objects. Experts argue over the capabilities and liabilities of the extended relational approach versus object-oriented, but conventional wisdom suggests that both approaches will coexist. The potential savings in development and maintenance of the object-oriented technology seems to be balanced by the widespread use of long-held relational-based applications [14:35-36].

An object-oriented database system is a persistent and shareable repository and manager of an object-oriented database -- itself a collection of objects defined by an object-oriented data model. A data model is a logical organization of real-world entities. Object-oriented concepts form a good basis for developing a rich data model for next-generation database applications. There is no standard data model on which to construct an object-oriented database language from which to program applications [14:36-40].

Won Kim proposes a set of rules to define object-oriented database systems. First, an object-oriented database system must provide all conventional database facilities. Second, an object-oriented database systems must allow the following [14:36-40]:

- Real world entities must be modeled as objects with a unique identifier for each object.
- Every object must have a state and a behavior (attributes).
- Objects may be grouped into classes according to attributes and methods.
- A class attribute's domain (type) may be a class.
- Classes must be organized into a rooted directed acyclic graph or hierarchy.
- An object's state and behavior can only be accessed or invoked from outside the object through an explicit message.

Kim maintains that these concepts form the basis of any viable object-oriented data model, and any database language that would evolve. The three components of a database language are data definition, data manipulation, and data control. The first component,

data definition allows a programmer to specify a database schema or framework. In an object-oriented database a specification will define the interfaces to object types as described in the object model. The ability to treat any real world entity as an object simplifies the user's view of the world. Object identifiers are important to object-oriented systems because objects consist of attributes with values, and the system (as well as object-oriented languages) assume objects exist in a large virtual memory. The class concept is important to object-oriented database systems because it captures semantic data-modeling concepts (relationships), it is the base on which queries may be formed, it enhances integrity by virtue of type checking, and allows sharing of attributes and methods [14:42].

The second database language component, data manipulation provides facilities to express queries and updates to a specified database. The object-oriented database must allow users to [15:33]:

- Create, update, and delete individual instances of a class.
- Fetch an object using the object's unique identifier, or a collection of objects rooted at a user-specified object (navigational fetch).
- Fetch a set of objects satisfying user specified search conditions (declarative fetch).
- Allow queries on a single class and queries on more than one class.

The third database language component, data control, provides integrity protection of the database and manages system resources. The data control must allow specification of transactions (commit and abort), semantic integrity control (class methods), authorization (limit access), and management of access methods [15:36].

According to John Joseph and associates, the five principle reasons that conventional databases are inadequate in serving the needs of the next generation applications are [13:46-47]:

1) Lack of expressive data modeling. Next-generation applications require the same level of expressive power as programming languages, such as complex data types (arrays, records, class definitions, and functions) or control structures (conditional clauses and procedure calls).

2) "Impedance mismatch" between programming languages and database systems. Impedance mismatch is the difference between the programming languages required to support differing data models and paradigms for object manipulation. This mismatch decreases application programmer productivity by requiring complex problems to be mapped to the conventional database. The mismatch also requires programmers to use two programming languages and paradigms in the two environments.

3) Interactive performance does not support next generation applications. The expense of a relational query to fetch a single, identified object is too high because it often requires burdensome overhead when compared with the simple offset addressing of an object-oriented database.

4) Lack of mechanisms to support long transactions. Long transactions require cooperative transactions, in contrast to the short duration of conventional database transactions locking very little data and involving infrequent locking conflicts.

5) Lack of mechanisms to support schema evolution and version management. By conventional thinking databases only have a single state -- the current state; there is no version management.

Conclusion. In conclusion, this literature review has provided an overview of the Object Modeling Technique, presented limitations of a relational database system, and

discussed the engineering features of object-oriented databases to support database system functions. All three of these areas are required for the successful analysis and re-engineering of the AFITSIS.

III. Analysis and Design

Introduction. This chapter presents the steps necessary to construct an analysis model and the process in building the subsequent design model for STARS as a proof of concept for this research. The approach includes the use of the analysis process presented with the Object Modeling Technique (OMT) as documented in *Object-Oriented Modeling and Design*, by Rumbaugh and associates, and pictured in Figure 3.1 [19:149]. To maintain the distinction between the analysis model and design model it is useful to remember the following general definitions. The analysis model includes information meaningful to the client of the system; it is an external view of the system. The design model is constructed for computer implementation; it must be efficient and practical to encode.

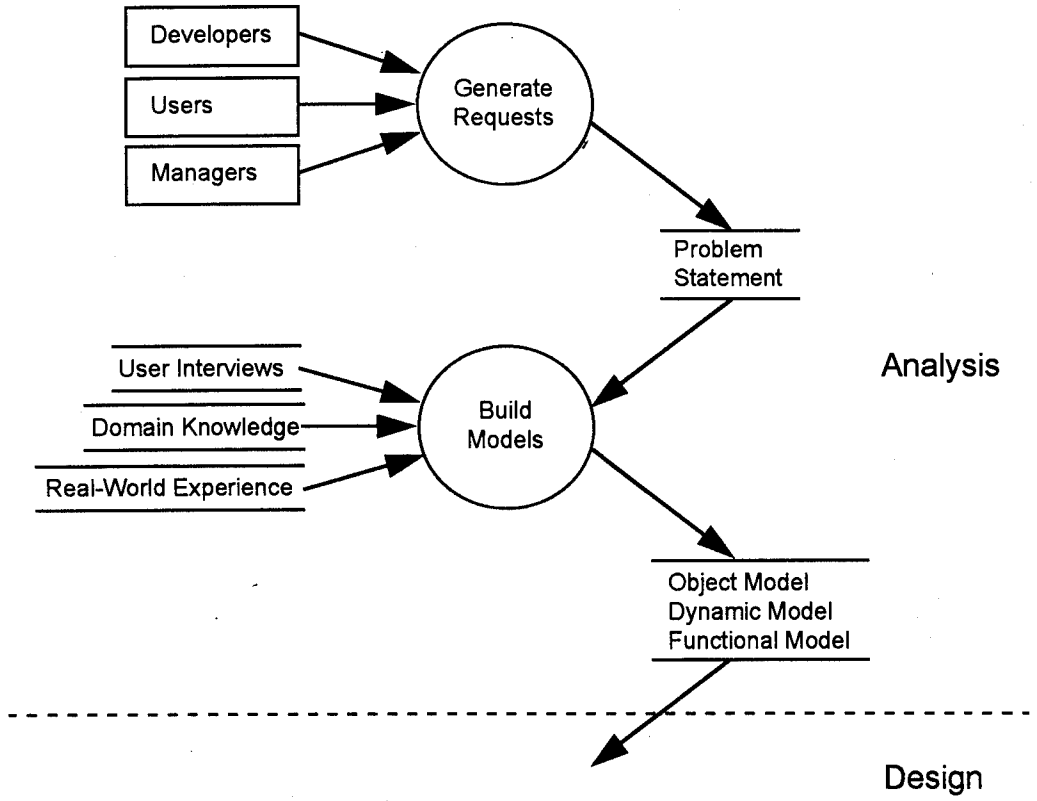


Figure 3.1 Overview of the analysis process (Rumbaugh and others, p.149).

Although many courses and research revolve around object-oriented programming languages and coding, Rumbaugh and associates emphasize from the beginning that OMT applies to a system's entire software development cycle. The OMT cycle begins with the analysis of a system problem statement and continues until the end of the software system's life-cycle. The results from the analysis phase is carried forward to system and object design. Finally, the results are used by software programmers to implement the final system. The key part of this whole process are the three models -- object, dynamic, and functional. Although developed at the beginning of the process from the problem statement, all three models undergo constant scrutiny and revision from start to finish and in effect become extremely valuable documentation, not only for the system designers, but also the maintainers late in the software's life-cycle.

Analysis. The OMT approach was followed in this paper and begins with the a simple problem statement upon which three models are constructed to capture the system from three related but differing viewpoints. The three models as introduced in chapter two are: the object model, the dynamic model, and the functional model. The *object model* captures the static, structural, or "data" aspects of the system. The *dynamic model* captures the behavioral, or "control" aspects of the system. The *functional model* captures the transformational, or "functional" aspect of a system. Systems developed under such an approach may rely heavily upon one or more aspects of the three models depending upon software requirements. For example, in the database portion of the STARS software where data persistence and manipulation are the main focus, the object model is the most important model. The object model shows which objects are responsible for what data (attributes) and how the different objects are related with one another (relationships). The object model is followed, in importance, by the dynamic model which shows concurrent access of distributed information, and may be used to

estimate transaction throughput. The functional model is less important because operations are usually predefined and tend to focus on creating, updating, and querying information [19:148-149].

The following sections outline the creation of the analysis model beginning with the problem statement, detailing construction of the object, dynamic, and functional models, and discussing the final analysis of the three models.

Problem Statement. The maintainers and users of the current system, along with development documentation, provided the requirements necessary to

> The Student Tracking and Registration System (STARS) should be designed for use by students, faculty, and administration personnel. The system shall distinguish between authorized and unauthorized users. Among authorized users operations shall be granted depending upon their authority. For example, faculty and administrative personnel will have privileges that a student will not.
> The database system will perform such typical operations as:
>
> a. creating data,
> b. storing persistent data,
> c. retrieving data,
> d. editing data, and
> e. deleting data.
>
> The database system will save information relevant to students, instructors, dependents of military members, and sponsors of research areas. The system will also track school and course registration data, such as course sections, class rosters, grades assigned, GPAs, etc. A user will log into the database application and be provided operation choices. Only those functions for which the user has authorization will be offered. Typical user queries will be provided. For example, the user should be able to view and output the courses scheduled for a particular quarter including instructors, times, days, and assigned room(s). Queries will be possible on a student's overall, planned, completed, and present schedules. Modifications shall be allowed on planned and current schedules by proper personnel. Modifications to course grades will only be allowed by authorized authority. Access to database information for purposes of printing or viewing should be allowed concurrently by any number of users. Access to data being created or modified should prevent other users from accessing and/or modifying the information at the same time. Historical records must be maintained for changes to certain database fields (i.e. SSAN, Name, Rank, Program, Graduation Date).

Figure 3.2 Student Tracking and Registration System Problem Statement

implement a system, as well as future requirements for anticipated growth. From this foundation a STARS problem statement was formed as shown in Figure 3.2.

Object Model. The object model describes the identity, relationships, attributes, and operations of objects in a system. An object model was constructed by [19:21-47]:

i) identifying objects and classes from the problem statement,

ii) identifying associations between objects,

iii) identifying attributes of the objects and links,

iv) verifying access paths exist for queries, and

v) refining the object model with numerous iterations.

Identify Object Classes. All relevant objects were identified from the database domain. Explicit object classes are the easiest to identify by extracting the nouns present in the overall requirements statement. Figure 3.3 lists the tentative object classes obtained from the requirements statement.

authorized users	students
unauthorized users	database system
data	personnel
records	instructors
students	grades
database system	courses
persistent data	rooms
instructors	authority
dependents	application
military members	operation
sponsors	functions
research areas	historical records
schedule	26 menus (main menu/25 submenus)
planned schedule	completed schedule
present schedule	

Figure 3.3 Candidate object classes

Additional implicit classes are not as easy to identify and must be obtained by knowledge of the problem domain. The implicit object classes such as forms, international students,

and departments were derived from interviews with the maintainers and development documentation.

Object classes were eliminated from the candidate list for the following reasons:

a) Redundant classes expressing the same information. Duplicate classes of *students*, *database system*, and *instructors* may be eliminated.

b) Irrelevant classes having nothing to do with the problem. *Personnel* does not adequately describe the object class and is replaced with *person*.

c) Vague classes too broad in scope. *Records*, *historical records*, *data*, and *persistent data* all describe the information we wish to maintain through the use of objects' specific attributes.

d) Classes that are attributes of another class. Grades are attributes of a *person* class or an attribute of the relationship between a *person* and a *course*.

e) Class is an operation of another class. *Functions* and *operations* are too general to implement and will become specific object functions or procedures during design and implementation.

f) Implementation construct. *Authority* will be decided during implementation; it could also be viewed as an attribute of a user of the application.

The result of identifying object classes is documented in a data dictionary identifying each class and describing its relationship to the problem being modeled. The data dictionary for the STARS object model is provided in Appendix A. Current object classes are depicted in Figure 3.4.

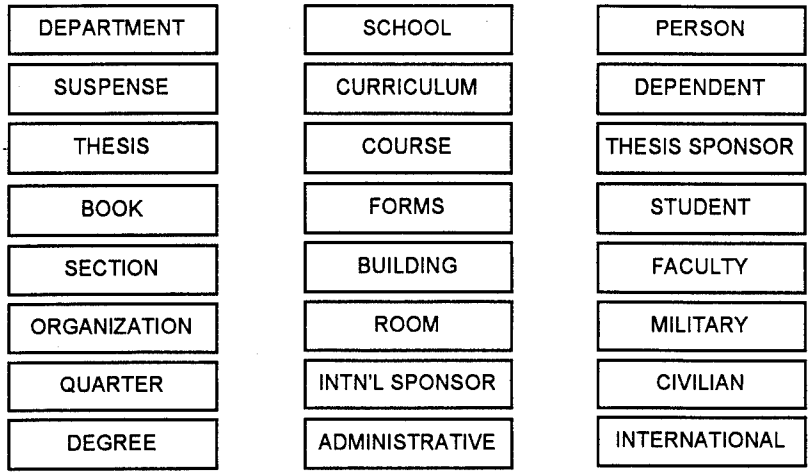


Figure 3.4 Current Object Classes

Identify Associations. Next, all dependencies were identified between two or more classes and documented as associations between classes. Figure 3.5 lists all candidate associations from the problem statement and problem domain. Explicit associations can usually be identified as verbs or verb phrases within the problem statement. Other associations were identified from user documentation or as part of the inherit problem domain.

Verb Phrases:

distinguish between authorized or unauthorized user
distinguish among :
1. Student User
2. Faculty User
3. Administrative User
query student schedules
modify schedules
printing or viewing by any number of users
limit user access for creating and modifying
perform authorized functions
maintain historical records
save information
provide menus
store persistent data
retrieve data
edit data
create data
delete data
log into application
modify grades
access database

Implicit Verb Phrases:

faculty may be either civilian or military persons
students may be either civilian or military persons
dependents are a type of person
international students are a type of person
faculty are a type of person
students are a type of person
sponsors are a type of person

Knowledge of Problem Domain:

faculty teach courses
faculty advise students academically
faculty advise thesis students
faculty are members of theses committees
sections are offered by quarter
curricula are made up of courses
students write theses
students take courses
courses require book(s)
sections meet in rooms
courses require prerequisites
courses are taught as sections
rooms are part of buildings

Figure 3.5 Candidate Associations

Identify Attributes. Attributes were identified by consulting the STARS Users Manual and a listing of the 151 relational tables in the current AFITSIS database system. Appendix B provides the menus and possible operations as depicted in the STARS Users Manual, and the relational tables with fields from the current system. As an example, Figure 3.6 is a current relational table in the AFIT database used by STARS to represent a Person. Attributes that could be used to represent characteristics of a person class could be social security account number, grade/rank, name prefix, first name, last name, middle initial, date of birth, etc.

Person (SSAN, Grade_Rank_Abbrev, Name_Prefix, Name_Suffix, First_Name, Last_Name, Middle_Initial, Birth_Date, Sex_Code, Race_Code, Marital_Status_Code, Religion_Code, Blue_Chip_Indicator, Aka_FName, Aka_LName, Prior_AFIT_Months, TAFMS_Date, Ethnic_Group_Code, Aero_Rating_Code, Manning_Code, DEROS_Date, Separation_Date, Commission_Code, Grade_Rank_Date, Citizenship00Country_Code, Department_Code, Duty_Title, Duty_Phone, Duty_Area_Code, Badge_Number, Branch_Service_Code, Login_Name, Input_Date, Duty_Phone_Ext)

Figure 3.6 Relational table for Person

The object classes, relationships, and attributes identified in the sections above were then arranged in the object diagrams pictured as part of the Data Dictionary at Appendix A. Figures 3.7 through 3.12 exhibit basic object modeling concepts (without object class attributes) used in development of the final object diagrams.

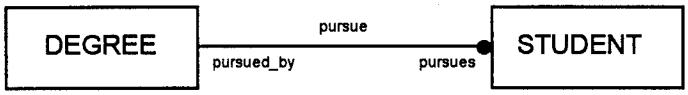


Figure 3.7 One-to-many Association

The one-to-many association in Figure 3.7 captures the relationship between *Degree* and *Student*. A degree is pursued by many students, or the inverse relationship of a student pursuing a degree.

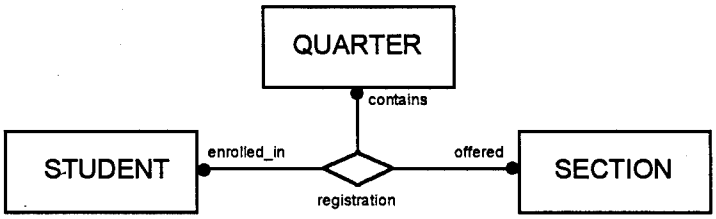


Figure 3.8 Ternary Association

The ternary association in Figure 3.8 shows the Student registration relationship with *Quarter* and *Section* object classes. A student may be enrolled in multiple quarters and multiple sections. A quarter will have many students registered for multiple sections. Finally, a section will be offered in specific quarter(s) and enroll a number of students.

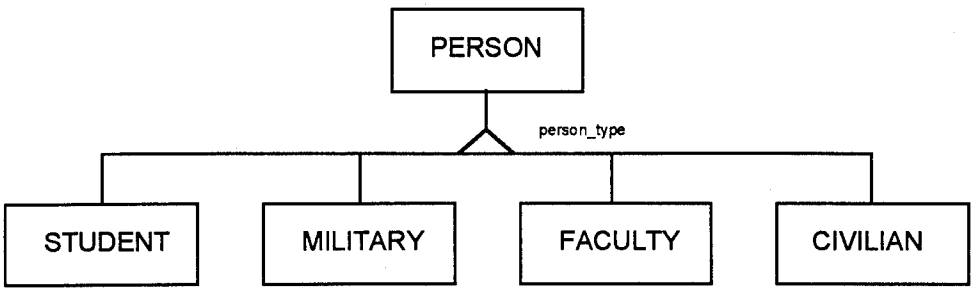


Figure 3.9 Inheritance

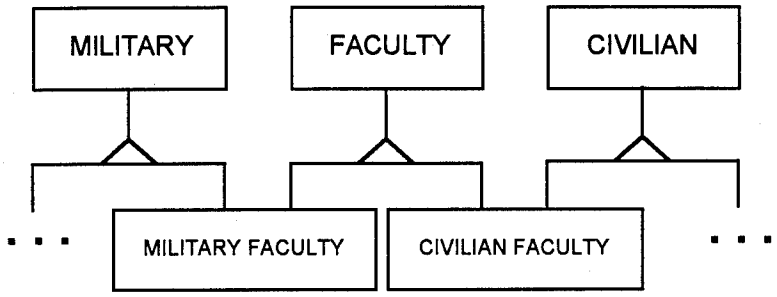


Figure 3.10 Multiple inheritance

The inheritance and multiple inheritance object diagrams shown in Figures 3.9 and 3.10 model the different types of people within the school.

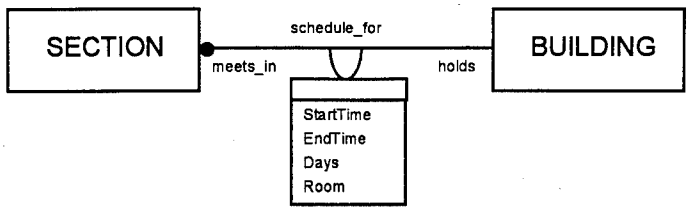


Figure 3.11 Association with link attribute

The link attributes, shown in Figure 3.11, on the *schedule_for* association between *Section* and *Building* captures attributes unique to the relationship between the two objects.

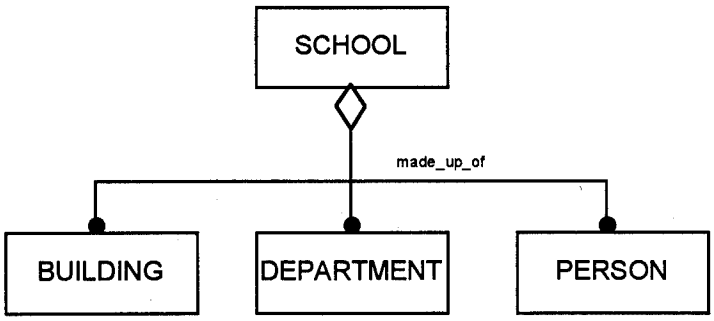


Figure 3.12 Aggregation

The aggregation modeled in Figure 3.12 depicts the *Building*, *Department*, and *Person* objects necessary to construct the school object.

Access Paths for Likely Queries. One way to verify that the object diagrams obtained thus far are accurate is to trace access paths through the object model to see if they yield sensible results. One would assume when a user to the STARS entered the application that they would have a particular interest in mind for the data. This would probably be creating, modifying, displaying, or deleting records or data. Paths for likely queries would be to obtain information from one of the objects (such as a student's GPA) or to be able to obtain relationships between objects (such as a list of

classes being taken by a student for a particular quarter). The object model in the STARS database contains a large number of possible queries depending upon the user and the interests in the data. An administrative user may wish to know how many students were enrolled per department for the last ten years. Registration personnel may be interested in whether all student records are complete and up to date. Student's may be interested in their schedules for past, present, and future quarters. Faculty members may be concerned with inputting the final grades for the quarter. No matter what the intent of the user, the application needs to access the database and provide the function requested by the user.

Because of the large number of queries possible, the implementation of the proof of concept for this thesis focuses on a user viewing a class roster and a student's schedule.

Refine Object Model. Refining the object model is a continuous process of iteration, not just during the object modeling phase, but at every stage of the analysis and design process. Object classes that are found to be needed during later stages of the analysis process need to be incorporated into the object model at that time. Complex object models can be grouped by classes into modules which represent some logical subset of the entire model.

Dynamic Model. The dynamic model shows the time-dependent behavior of a system and objects within it. A dynamic model for STARS was constructed by [17:84-113]:

i) preparing scenarios of typical interaction sequences,

ii) identifying events between objects,

iii) preparing an event flow diagram for the system,

iv) building state diagrams and state transition tables, and

v) matching events between objects to verify consistency.

Scenarios. Scenarios were created to better understand typical exchanges between a user and the system. Two scenarios for the student tracking and

registration system are presented. First, a faculty user scenario (Figure 3.13) describes the steps an instructor uses to print out a class roster. Figure 3.14 depicts the second scenario of an administrative user creating a new instance of a student. (Note: the second scenario only considers the steps changed from the first scenario.)

The user starts Window application.
The application requests user name and password; the user enters name and password.
The application verifies user name and password with database system; system recognizes authorized user (as type faculty) and notifies application.
The application presents faculty user with menu of operations (view student schedule, print student schedule, view faculty schedule, print faculty schedule, view class roster, print class roster, ...); the user selects print class roster.
The application asks user to identify the class by section and quarter; the user identifies class.
The application sends system roster request.
The system passes application the requested class roster.
The application prints class roster.
The application presents user with menu of operations; user selects exit.
The application terminates.

Figure 3.13 Faculty User Scenario

The first three steps are the same as Figure 3.13 above.
Application presents administrative user with menu of operations (create student, create faculty, modify student, modify faculty, delete student, delete faculty, ...); the user selects create new student.
The application sends system create new student request.
The system presents application with a blank form to be filled out by user.
The user fills in form (student's last name, first name, middle initial, SSAN, ...)
The user submits data.
The application passes data to system.
The application presents administrative user with menu of operations; the user selects exit.
The application terminates.

Figure 3.14 Administrative User Scenario

Events Between Objects. Events to and from a typical user and/or any external devices were identified from the two scenarios in Figures 3.13 and 3.14. Event traces were constructed from the scenarios described exhibiting the events that arise between the objects of different classes. The user in Figures 3.15 and 3.16 represents a person being queried for information and responding to the requests. Although not shown on these event traces, error conditions and unusual events need also to be considered when listing all possible events between objects.

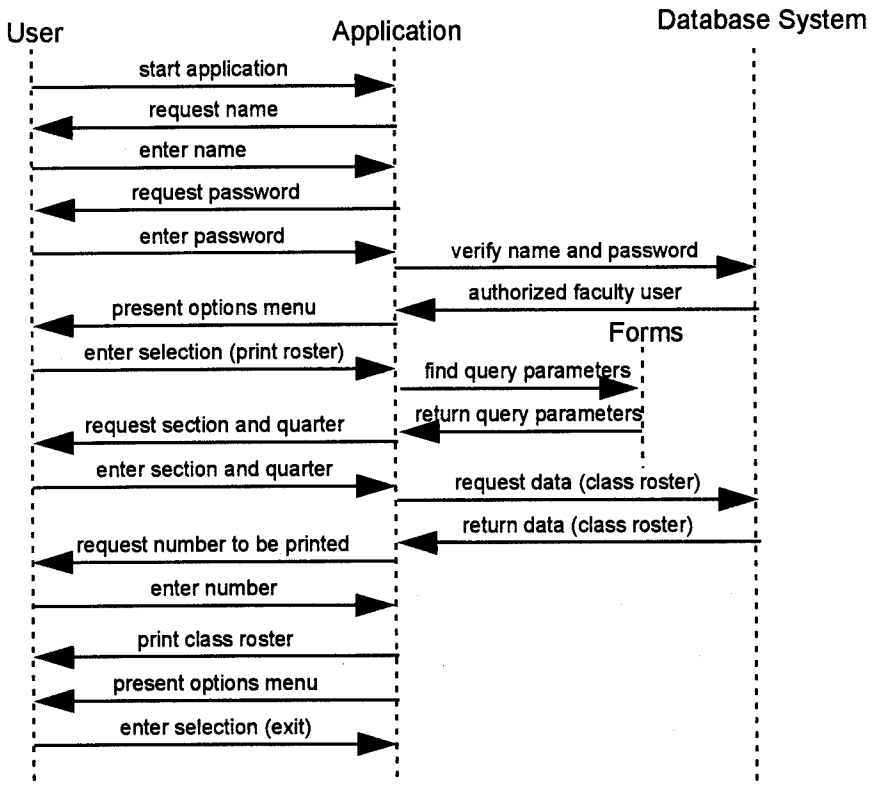


Figure 3.15 Event trace for faculty user scenario

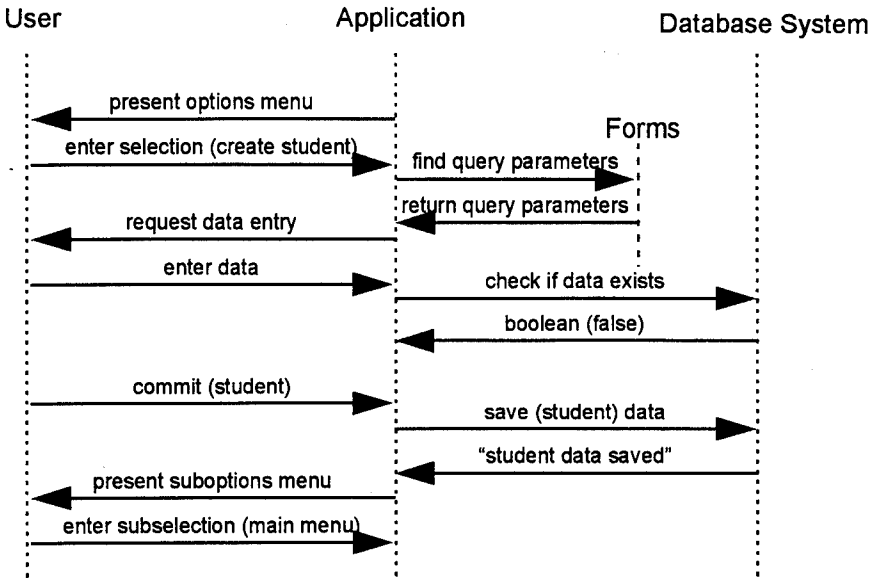


Figure 3.16 Event trace for administrative user scenario

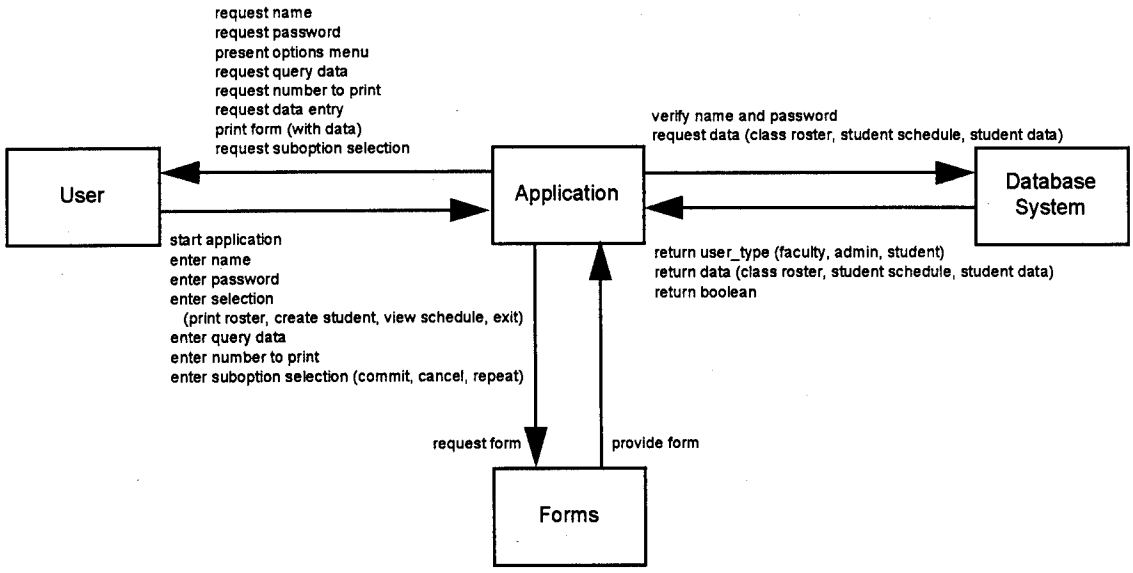


Figure 3.17 Event flow diagram for scenarios

Event Flow Diagrams. The event flow diagram in Figure 3.17 summarizes the events between classes without regard to sequencing. The event flow

diagram shows the information flow and serves as a dynamic counterpart to an object diagram.

State Diagrams. The next step in dynamic modeling is constructing state diagrams for each object class with nontrivial dynamic behavior showing the events that the object receives and sends. Every scenario corresponds to a path through the state diagram. A single object was considered and arranged with arrows labeling the input and output events from one state to the next. The intervals between two events can be considered a state. Merging states and events from other scenarios provides alternative paths, and traversal loops that allow scenarios to have differing outcomes when traversed in different permutations.

The state diagram process began by modeling the application object in its simplest form -- either *Idle* (following application start-up) or *In_Use* (beginning when the user logs-in and ending with either an invalid login or program exit). At this high level, the application is either *Idle* or *In_Use*. Following application start-up a user could enter a name, fail user authorization, in which case the application will return to the Idle state, or use the application and then exit.

All four of the events are shown in the state diagram of Figure 3.18 below. *StartApplication* begins the diagram (represented with the single solid dot) and the [EXIT] criteria ends the flow through the state diagram (represented with the circled solid dot).

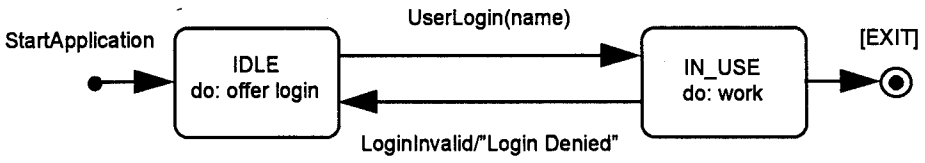


Figure 3.18 Application state diagram

The *In_Use* state in Figure 3.18 can be modeled with three substates identified from the event trace diagrams (Figures 3.15 and 3.16). Figure 3.19 shows the three substates (*Process Login*, *Present Options Menu*, and *Process Selection*) and the events triggering movement from one state to the next.

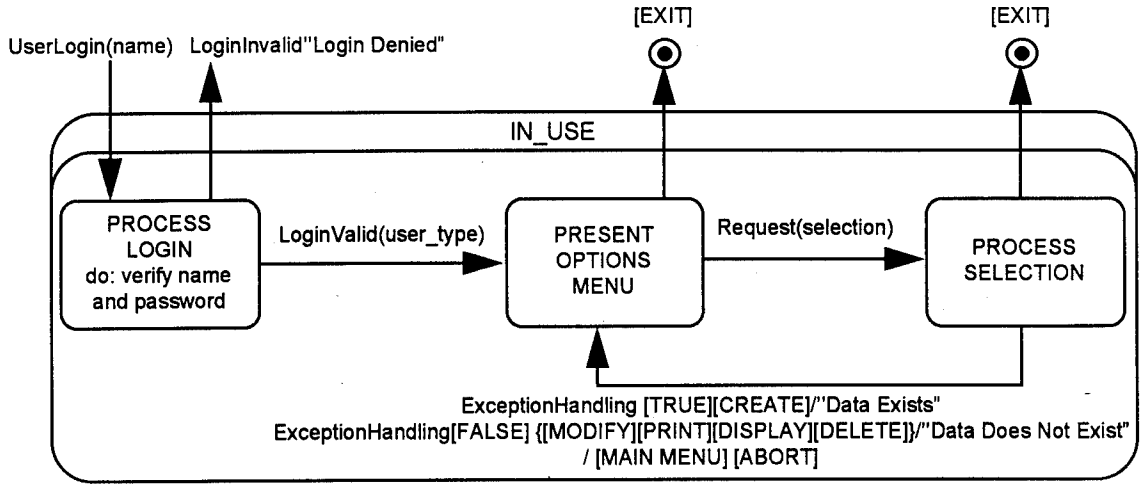


Figure 3.19 Substates of In_Use state

Figure 3.20 shows the *Present Options Menu* state as a dependent state on the guard variable *user_type*. The type of user identified during the *Process Login* state will dictate the menu options offered to the user. The three different types of users, administrative, faculty, and student, will have an assortment of options, some of which may or may not be the same as those offered to another type of user.

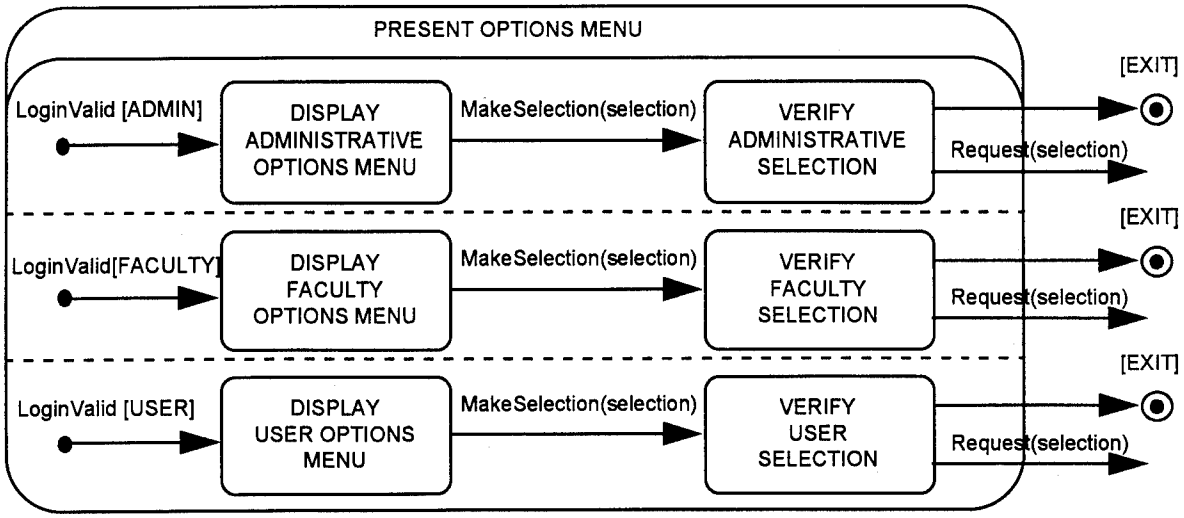


Figure 3.20 Present Options Menu state as dependent state

The modeling of the menus and submenus presents an interesting relationship between the two types of menus. A submenu, in general, is a lower tiered menu from the main menu. The main menu will contain a list of options from which a user selects. Menus may have a title and a list of options from which to make a selection. The first option of the menu will be the default selection. Only those options for which the user has authorization will be displayed as menu options.

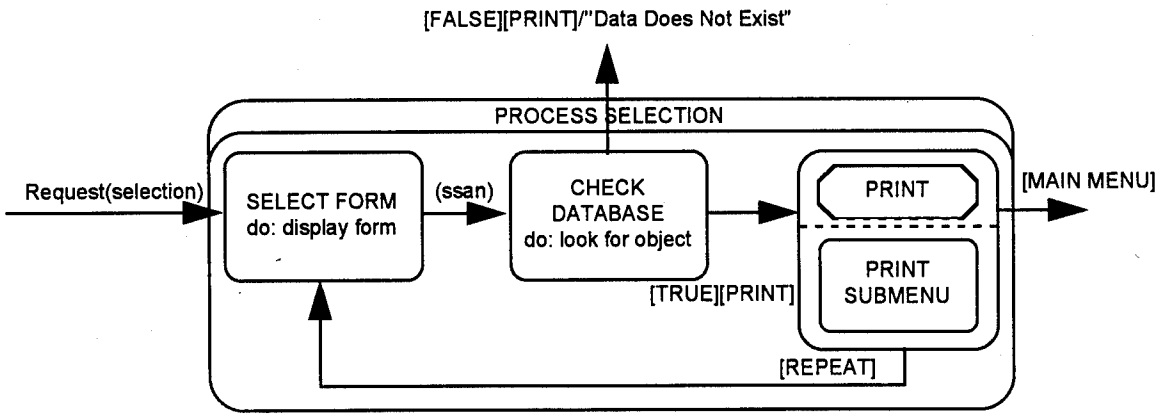


Figure 3.21 Partial substates for Process Selection state

The Process Selection substate shown in Figure 3.21 is further broken down into states required to handle the Print class roster request made by the faculty user in the earlier scenario (Figure 3.13). The substates for the Process Selection state consist of finding the type of form involved with the users request, using query data (in this case a person's SSAN) to see if the objects exist, and if the answer is true then the request is executed. If the answer is false then the user is notified and the application waits for another request.

A form is a screen display of a printed document. It may be in a fill-in-the-blank format and designed to a level where the user need not have any programming skills. The purpose of the form is to organize and display database information in a manner that is easily understandable and recognizable to the user.

Although not mentioned by Rumbaugh and others in their modeling technique, a state transition table provides a simple way of summarizing data provided in state diagrams. It also allows personnel responsible for the final analysis models and design to check if all entrance and exit criteria between states have been satisfied, as well as checking possible error conditions. Consider the *Idle* and *In_Use* states from Figures 3.18 and 3.19. When the event *UserLogin* occurs the parameter *name* is passed to the next state *In_Use*. The *Process Login* state (first substate of *In_Use*) then becomes the current state at which time one of four events could occur. The login could be invalid, in which case the control is passed back to the *Idle* state, or one of three types of *LoginValid* events could be triggered depending upon the *user_type* guard condition. A partial state transition table (Table 3.1) is shown below to coincide with the state diagrams just discussed. The complete set of state diagrams and state transition tables is provided in Appendix A.

Table 3.1 Partial state transition table for Application

Current State	Event	Parameters	Guard	Next State	Action
Idle	UserLogin	name		Process Login	
Process Login	LoginValid		ADMIN	Display Admin Menu	
Process Login	LoginValid		FACULTY	Display Faculty Menu	
Process Login	LoginValid		STUDENT	Display Student Menu	
Process Login	LoginInvalid			Idle	"Login Denied"

Additional substates for the *Present Options Menu* and *Process Selection* substates are presented as part of the Data Dictionary. States not considered in either the faculty or administrative scenarios are essential to the proper execution of a STARS application. These states have been combined with Figure 3.21 resulting in a final state diagram for the *Process Selection* state and complete state transition table shown in Appendix A.

Verify Consistency. Now that the state diagrams are complete we can check for consistency and completeness at the system level by following events through the state diagrams. There should be paths from initialization (StartApplication) to the termination of the state diagram (EXIT). Events should have both a sender and receiver. Corresponding events on different state diagrams should be consistent. The state transition tables aid the designer in this verification by tracking events, parameters, guard conditions, and actions from one state to another [12].

Functional Model. The functional model diagrams the flow of data among processes with data flow diagrams. The functional model complements the dynamic model in specifying the system's behavior. A functional model was constructed by:

i) identifying input and output variables,

ii) drawing data flow diagrams (DFD) with functional dependencies,

iii) describing functions,

iv) identifying constraints, and

v) specifying optimization criteria.

Input and Output Variables. Input and output variables are parameters of events between the system and the outside world. Figure 3.22 shows the input and output variables for the STARS application. Since all interactions between STARS and the outside world pass through the application, all input and output values are parameters of application events. Input events that only affect the flow of control, such as *Display*, *Print*, *Abort*, or *Commit*, do not supply input values.

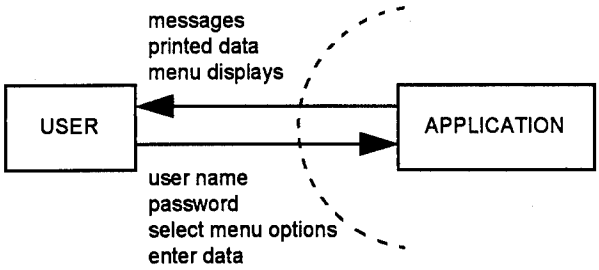


Figure 3.22 Input and output values for application

Data Flow Diagrams (DFD). Data flow diagrams show how each output value is computed from input values. The DFDs that follow were constructed in layers. Figure 3.23 shows the top-level (level 0) DFD for the STARS application. User represents an external object that supplies and consumes input and output values, respectively. Data flows are represented by the labeled arrows, data stores by the 'sandwiched' labels, and processes by labeled ovals. From this top-level diagram each non-trivial process may be expanded recursively into lower level DFDs until all processes are trivial. For example, the *perform action process* from Figure 3.23 is expanded into three processes, *validate user login*, *offer menu options*, and *handle selection*, as

exhibited in Figure 3.24. This level 1 DFD in turn is expanded into three level 2 DFDs. The first process, *validate user login* is shown in Figure 3.25. The remaining DFDs are contained in the Data Dictionary.

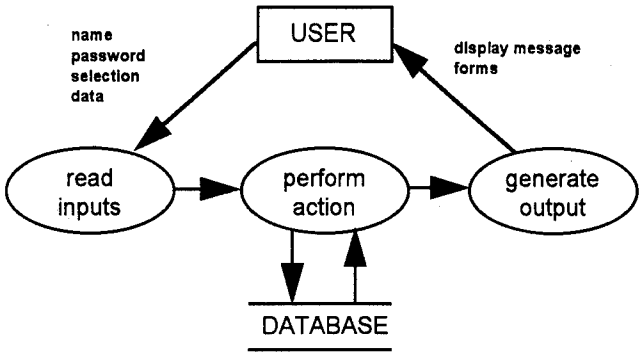


Figure 3.23 STARS Application Level 0 DFD

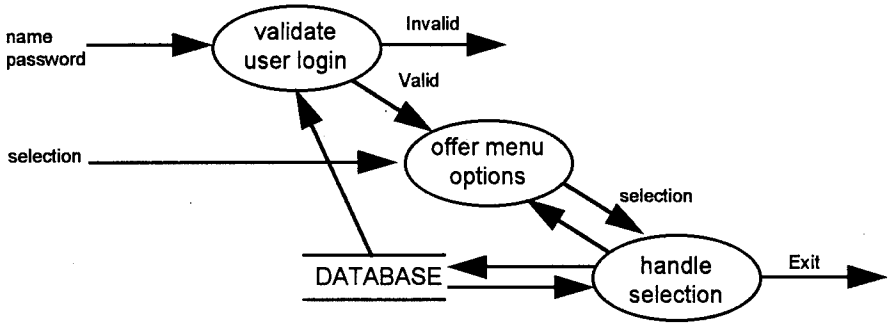


Figure 3.24 Perform Action Level 1 DFD

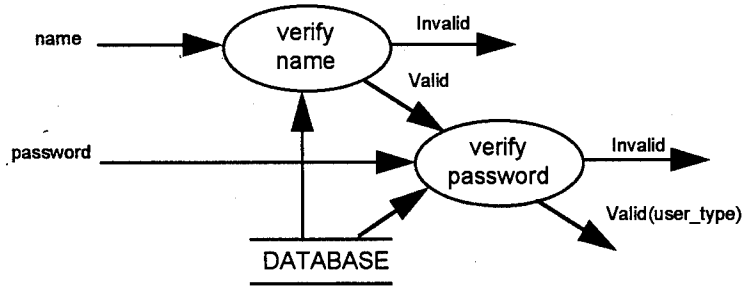


Figure 3.25 Validate User Login Level 2 DFD

operations. Higher-level processes may also be considered operations, although optimization may allow for a varying implementation. The specifications for non-trivial operations are contained in the Data Dictionary. Non-trivial operations can be divided into three categories: queries, actions, and activities. The function description for the *validate user login* process is outlined in Figure 3.26. Note the process relies on two variables, *name* and *password* as input, and provides *messages* and a *user_type* as output.

Validate User Login (name, password) → user_type, messages

If the name entered matches a valid user then

allow password verification

If the password entered matches a valid password then

send a message notifying user of valid access

If name or password entered does not match valid user then

send message notifying user of denied access

Figure 3.26 Function description for Validate User Login process

Constraints. Constraints show relationships between two objects at the same time or between different values of the same object at different times. Constraints may appear in all three analysis models. Preconditions on functions are constraints input values must satisfy, and post conditions are constraints that output values are guaranteed to hold. The conditions under which constraints must hold are stated in brackets.

Optimization Criteria. Potential optimizing criteria for a database system are to minimize the time objects are locked, or if needed the entire database is

locked. Another criteria is to maximize the use of functions and procedures by taking advantage of superclass-subclass relationships, and multiple inheritance.

Final Analysis. When the analysis is complete, the models should reflect the requirements expressed in the problem statement and those requested by the user. In an effort to provide a good final design the analysis process is wrapped up (but may continue through the software's life-cycle) by adding operations to the object model and verifying the models meet requirements.

Add Functional Model Operations. The defining of operations for objects is an open-ended process, but it is useful to try to identify potentially useful operations and summarize them on the object model. The operations are obtained from many of the diagrams and models we have already constructed. For example, the object model implies reading and writing attribute values and association links (such as *get_attribute* and *set_attribute*). Events sent to objects correspond to an operation on an object. Actions and activities in the state diagram may be functions (such as *Process Login* or *Verify Selection*). Each function in the DFD corresponds to an operation on an object or several objects.

After organizing all the operations on the object model, examine it for duplicate operations or variations of a single operation. Sometimes an object class will suggest operations that should be included for possible use in other problems (i.e. initialization or finalization). These should be added. The final object diagrams with operations are shown in Appendix A.

Compare Models with Problem Statement. One way to judge a design for soundness is to compare the models to the problem statement. The models should capture most requirements, and requirements involved with performance constraints should clearly be stated in the optimization criteria.

Develop More Detailed Scenarios. Another way of verifying correctness is by going back to the user, or consulting with an application domain expert. These activities amount to testing the domain limits by developing more detailed scenarios and verify that the models handle varying static, dynamic, and functional conditions. Additional scenarios are provided in Appendix A.

System Design. In *Object-Oriented Modeling and Design*, Rumbaugh and associates present an ideal life cycle for a database application as [19:367]:

1. Design the application.
2. Devise an architecture for coupling the application to a database.
3. Select a specific DBMS platform.
4. Design the database. Write DBMS code to set up the proper database structures.
5. Write programming language code to provide user interface, validate data, and perform computations.
6. Populate the database with information.
7. Run the application.

For the purpose of this research the first four of the seven stages of the application's life-cycle are of interest. Since we are not implementing executable code the last three stages are of little interest to us and are presented to provide for a complete life-cycle process. The first of the seven stages is captured as a result of OMT analysis effort -- the three models forming the framework of the design. The architecture chosen to couple the application to a database is transparent, as will be shown in the upcoming section on mapping object models to an OODBMS. This is contrasted with the more traditional approach of the table architecture supported by relational databases. The DBMS platform selected is one proposed as an ODBMS standard in R.G.G. Cattell and

others' book, *The Object Database Standard* [5]. Finally, designing the database is discussed in the following subsections and the *Implementation* chapter.

When using an existing database management system the main concern is to construct an object model and decide upon the transactions that must be considered atomic by the system. Steps in designing a transaction manager are [19:216]:

1. Map the object model directly into the database.
2. Determine which resources cannot be shared (units of concurrency).
3. Determine the set of resources that must be accessed together during a transaction (unit of transaction).
4. Design concurrency control for transactions.

The steps presented for designing a transaction manager deal with making sure that the integrity and consistency of stored data is maintained. This transaction management is atomic, which means that transactions either happen as a whole or do not happen. This research does not discuss the implementation of transaction control, although it is discussed in ODMG-93 and would be imperative to operating any information system that has multiple users and is expected to share data. The first step for the transaction manager is to map the object model into the DBMS. This is discussed in the next subsection. The remaining steps are not discussed and are provided to keep a complete representation of the process to be followed.

Object Model Mapped to Database. A relational database logically appears as a collection of tables. In the design of a relational database application it is necessary for the designer to map an object model (or entity-relationship diagram) into a table model. The programmer then takes this table model and implements it in a relational database language. SQL is used for the examples that follow since it the most widely used relational database language.

In contrast to this three-level approach of mapping design to implementation, object-oriented databases offer the potential of doing away with the table level of mapping and allowing the designer/implementor to map directly from the object model to implementation. Figure 3.27 summarizes this concept. The impact of this change will aid both developers and maintainers because software will be more tightly coupled to design and maintenance documentation.

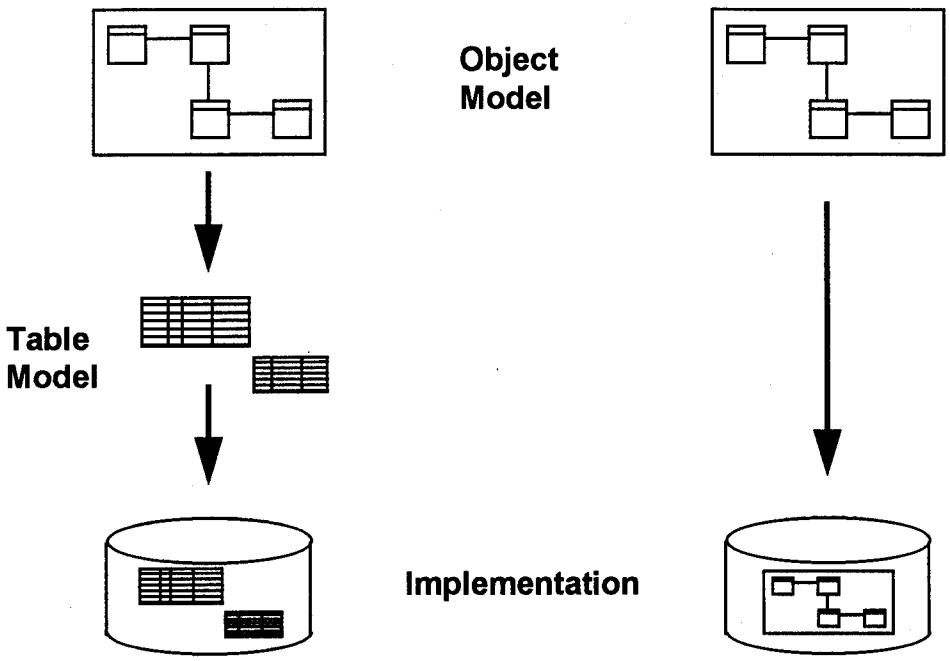


Figure 3.27 Comparison of Database Management System Architectures

In the book *Object-Oriented Design* [9:82-83], Coad and Yourdon describe this mapping from the object model to implementation as "a seamless view of objects; translating between storage data structures and program data structures is no longer needed" because object-oriented DBMS will take care of saving and restoring the objects. Designers and programmers will rely upon the ability of object-oriented DBMS to handle this type of activity and their work will involve mapping the object model into a feasible

design. Mapping basic modeling concepts to an implementation is relatively straightforward and easy to visualize. An example of mapping an object class to implementation is discussed for the *Person* object class below. Additional comparisons (i.e. associations, inheritance, multiple inheritance) are contained in Appendix A.

Advanced modeling concepts are more difficult to map to object-oriented databases. The two advanced modeling concept examples presented are for a ternary association and an association with link attributes. The following sections illustrate the differences experienced in mapping an object model into a relational database versus an object oriented database. In the examples, SQL and OQL (proposed in R.G.G. Cattell's book *The Object Database Standard: ODMG-93* [5]) are used to show a pseudo-code implementation.

Mapping Object Classes. The object class is the basic building block of object oriented design. Figure 3.28 illustrates the steps necessary to implement the *Person* object class in a relational database. First, the class *Person* is mapped to the Person table where a unique key is assigned for handling activities. Attributes of the class are assigned fields with data types defined by the domain, and required input constraints are defined by the predefined 'Null' constant. From this table the person class is implemented in a relational database with the SQL code shown.

Object Model

Person
Last Name
First Name
Middle Initial
SSAN

Table Model

Person Table

Attribute name	Nulls	Domain
person-ID	N	ID
last-name	N	name
middle-initial	Y	char
first-name	Y	name
SSAN	N	SSAN

Candidate Key: (person-ID)
Primary Key: (person-ID)
Frequently accessed: (person-ID) (SSAN) (last-name)

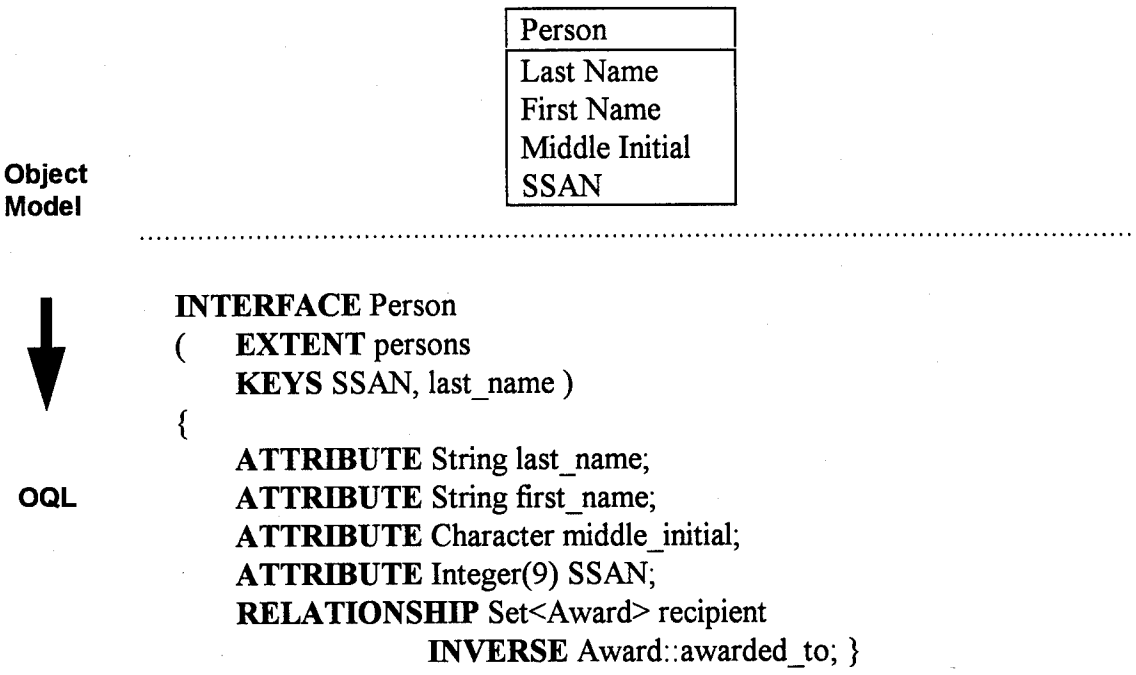
SQL Code

```
CREATE TABLE Person
    ( person-ID      ID          not null,
      last-name      char(30)    not null,
      middle-initial char                ,
      first-name     char(30)            ,
      SSAN           integer(9)  not null,
    PRIMARY KEY (person-ID) );
CREATE SECONDARY INDEX Person-index-name
      ON Person (last-name)
CREATE SECONDARY INDEX Person-index-name
    ON Person(SSAN)
```

Figure 3.28 Mapping object class to relational database

In comparison to the relational SQL implementation required to prepare the database for data input, the object-oriented database is more direct (object model to implementation) and easier to understand (no table concepts to interpret), as shown in Figure 3.29. The *Person* Object Identifiers (OID) are assigned with the key declaration, and attributes not assigned default values require input upon creation. The keys in Figure

3.29 are *SSAN* and *last_name* which allow an instance of the *Person* object to be uniquely identified. The key designator also requires that the attributes listed be provided values upon an object instance creation, just as the 'not null' did in the relational example. In contrast to the relational depiction in Figure 3.28 where likely access paths are identified through the creation of indexes access through objects will take place through the associations modeled in the OMT object diagrams. The relationship identifier *recipient* is shown with an inverse identifier of *awarded_to* shows a likely access path between the two objects *Person* and *Award.*

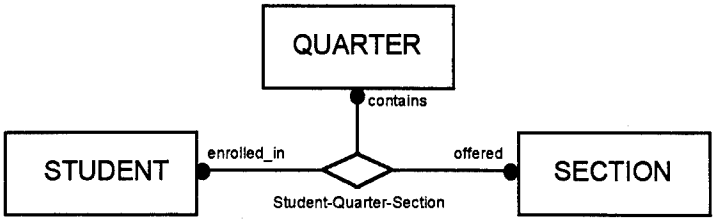


Note: Attributes may be assigned default values.

Figure 3.29 Mapping object class to object-oriented database

Mapping Ternary Associations. Mapping ternary associations into a relational database is handled by creating a table of table keys for the ternary association and any additional fields for link attributes. An example of this mapping is shown in Figure 3.30. A primary key is created on the tuple (student-ID, quarter-ID, section-ID). Additional access to the tables is provided through each foreign key on student, quarter, or section.

Object Model



candidate key(student-ID, quarter-ID, section-ID)

Student, Quarter, and Section Tables created similar to Figure 3.28

Table Model

Student-Quarter-Section Table

Attribute name	Nulls	Domain
student-ID	N	ID
quarter-ID	N	ID
section-ID	N	ID

Candidate key: (student-ID, quarter-ID, section-ID)
Primary key: (student-ID, quarter-ID, section-ID)
Frequently accessed: (student-ID)(quarter-ID)(section-ID)

SQL Code

```
CREATE TABLE Student
                ( student-ID     ID      not null,
                        attributes,
                PRIMARY KEY (student-ID) );
Similar CREATE TABLES for Quarter and Section

CREATE TABLE Student-Quarter-Section-ternary
                ( student-ID     ID      not null,
                  quarter-ID     ID      not null,
                  section-ID     ID      not null,
    PRIMARY KEY (student-ID, quarter-ID, section-ID)
    FOREIGN KEY (student-ID) REFERENCES Student,
    FOREIGN KEY (quarter-ID) REFERENCES Quarter,
    FOREIGN KEY (section-ID) REFERENCES Section );
```

Figure 3.30 Mapping ternary association to relational database

In the ODMG-93 standard [5] the authors specifically define relationships as types between object types. The ODMG-93 object model can support only binary relationships (no *n*-ary relationships). An object model allowing only binary relationships forces designers to break *n*-ary relationships into some binary equivalent. As pointed out in *Object Modeling and Design*, the majority of *n*-ary relationships can successfully be modeled as binary relationships[19:159].

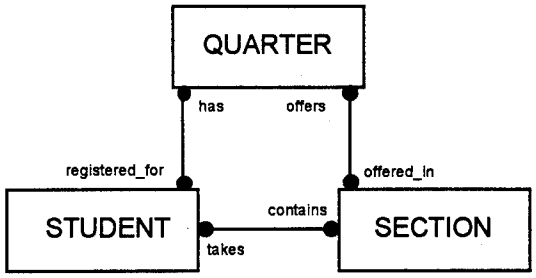


Figure 3.31 Ternary association with binary relationships

Figure 3.31 demonstrates one approach to modeling the ternary association with binary relationships. The difficulty in implementing this approach is that the enforcement of rules managing the access of objects via relationships is left to the implementor. Take as an example, a known student OID, and a requested output of all sections taken in all quarters for the given student. This output requires the traversal of the many-to-many relationship between *Student* and *Quarter* to obtain the set of quarters for which the student is enrolled. Each quarter results in another many-to-many relationship between *Quarter* and *Section*. Traversing this relationship for each quarter in the set produces the set of course sections offered in those quarters. The implementor would then use the many-to-many *Student-Section* relationship to obtain the set of course sections for which the student is taking courses. This set would then need to be logically intersected with the

previous set of course sections to obtain the common subset of courses. This subset could then be further divided into smaller subsets by using the *offers* relationship and obtaining the sets of sections taken by the student for each quarter in which they were enrolled.

Implementing this object model requires each of the object classes *Student, Quarter,* and *Section* to define two binary relationships; one each to the other two object classes. The corresponding Object Data Language for the *Section* object is shown in Figure 3.32. The operations defined by the user would need to take into account both the *offered_in* and *enrolls* relationships as well as the existence of any link attributes embedded in one of the other objects.

```
interface Section
(       extent sections )
{
        attribute Character symbol;
        relationship Set<Quarter> offered_in
                inverse Quarter::offers;
        relationship Set<Student> enrolls
                inverse Student::enrolled_in;
        operations
};
```

Figure 3.32 ODL for ternary association modeled with binary relationships

Another approach to modeling the ternary association is to abstract the relationship as an object consisting of inverse pointers to the objects forming the association. Figure 3.33 shows this modeling approach with each object class, *Student, Quarter,* and *Section* having a one-to-many relationship with the object *Registration*, thus forming the ternary relationship among the three objects. Figure 3.34 presents the corresponding Object Data Language for the *Section* and *Registration* objects from Figure 3.33. As shown in the figures, the relationship object *Registration* will contain the

operations permissible for the ternary relationship among the objects *Student*, *Quarter*, and *Section*.

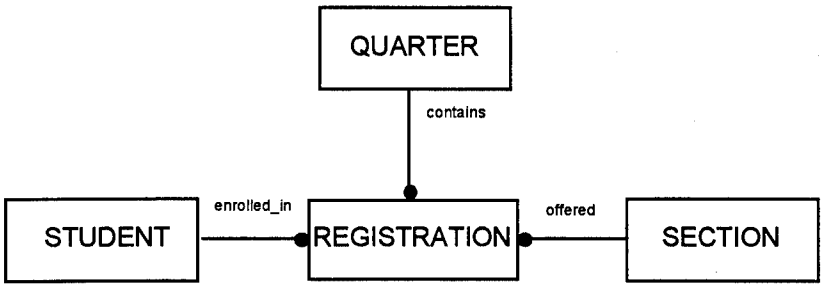


Figure 3.33 Abstracting the ternary relationship as an object

```
interface Section
(     extent sections )
{
      attribute Character symbol;
      relationship Set<Registration> offered_in inverse Registration::offers;
      operations
}
```

Similar implementation for Student and Section

```
interface Registration
(     extent registration )
{
      link attributes as required
      relationship Set<Section> offers inverse Section::offered_in;
      relationship Set<Student> enrolled inverse Student::enrolled_in;
      relationship Set<Quarter> contains inverse Quarter::contained_in
      operations
}
```

Figure 3.34 ODL for abstract ternary relationship

Considering this example is one of six scenarios for acquiring data from this ternary association it is easy to see why making provisions for ternary associations within

an OODBMS would be a useful and powerful addition. An additional object modeling constraint arises when a link attribute to the ternary association is considered. The link attribute modeling notation causes similar problems as previously discussed because the embedding of link attributes to one of the three classes requires the implementor to provide access to the link attributes from any of the other object classes.

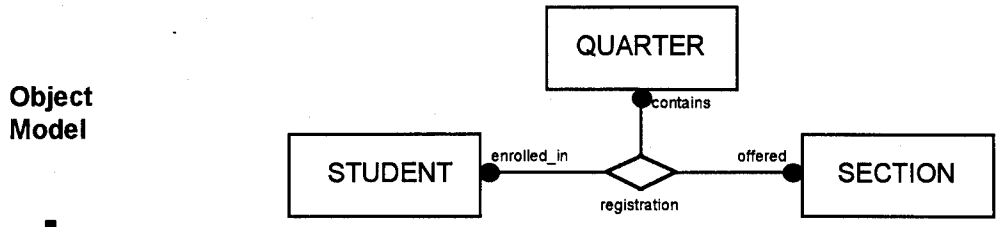
Rumbaugh and associates also note the fact that certain ternary relationships cannot be successfully broken into binary relationships without losing some information captured in the model [19:159]. The *Student-Quarter-Section* relationship is one such example. Considering this observation and the added complexity forced upon the database user in trying to manage ternary relationships, a more effective solution would be to extend the ODMG-93 standard [5:59-64] to allow ternary associations and manage the traversals among the three objects and any link attributes.

The proposed Backus Naur Form (BNF) for an *n*-ary relationship specification follows:

<ternary_rel_declaration>	::=	**ternary_relationship** <relationships_list> [**traverse** <identifier_list>] [**{order_by** <attribute_list>**}**]
<relationships_list>	::=	<target_of_path> <identifier> | <target_of_path> <identifier>, <relationships_list>
<target_of_path>	::=	<identifier> | <relationship_collection_type><<identifier>>
<identifier_list>	::=	<identifier> | <identifier> :: <identifier> | <identifier> :: <identifier>, <identifier_list>
<attribute_list>	::=	<scoped_name>, <attribute_list>

This extension allows the ternary relationship *Student-Quarter-Section* to be defined and added to the objects' specifications. As an example, Figure 3.35 shows the ternary association with the proposed BNF specification applied to the *Section* type's interface

specification. Similar specifications would be created for the *Student* and *Quarter* object classes.



OQL

```
interface Section
(       extent sections
        keys symbol ): persistent
{
        attribute String symbol
        ternary_relationship Set<Students> class_roster,
                             Set<Quarters> offered_in
                traverse Set<Quarters> :: schedule_of_courses
                traverse Set<Students> :: registered_for  };
```

Figure 3.35 Mapping ternary association to object-oriented database

This approach to capturing ternary associations follows the guidelines set forth in ODMG-93 for relationships (except for no *n*-ary relationships) [4:23,45,54]:

- relationship types are defined between (mutable) object types
- traversal paths are defined between (mutable) object types
- traversal names are defined for each direction of traversal
- traversal names are declared within the interface definitions of the object type
- relationships maintain referential integrity
- relationship instances do not have OIDs

This proposed extension would group an objects' relationships together so that operations affecting the *Student-Quarter-Section* ternary association could be handled by the ODMS. However, it does not solve the problem of implementing a ternary association with link

attributes. Since relationships do not have OIDs, link attributes cannot be stored in a relationship itself. The designer or implementor is thus forced to implement link attributes as part of one of the objects in the relationships.

Mapping associations with link attributes. A second proposed extension to ODMG-93 would relax three guidelines of the standard for implementing a relationship (besides no *n*-ary relationships) and allow for link attributes on all relationships. The guidelines that would be broken are:

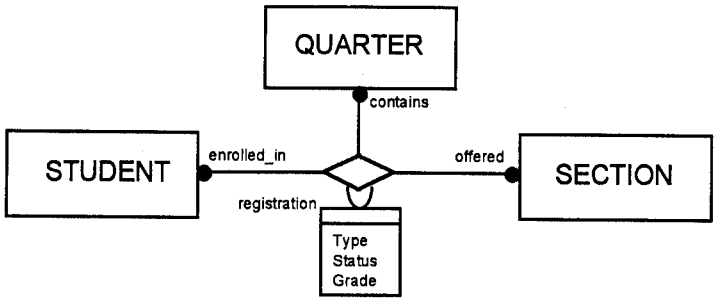
- traversal paths are defined between (mutable) object types
- traversal names are defined for each direction of traversal, and
- relationship instances do not have OIDs

Conceptually, what this second approach proposes is to treat relationships as an object, in which object pointers involved in the relationship are stored along with link attributes. Permissible operations on the association or link attributes would reside with the relationship object. The BNF for this second approach would be:

<link_relationship_declaration> ::= **link_relationship** <relationship_identifier>

As an example, the *Student-Quarter-Section* ternary relationship would now be implemented as shown in Figure 3.36. This approach to implementing a ternary relationship allows the implementor to associate link attributes with the relationship in which they logically occur.

Object Model



OQL

```
interface Section
(       extent sections
        keys symbol ): persistent
{
        attribute String symbol
        link_relationship Registration  };
```

Similar interface specifications for Quarter and Student

```
interface Registration
(           keys student-OID
            quarter OID
            section-OID ): persistent

    relationship Set <Students>
    relationship Set <Quarters>
    relationship Set <Sections>
    attribute Enumeration type
    attribute Enumeration status
    attribute Enumeration grade  };
```

Figure 3.36 Mapping association with link attribute to object database

Summary. This chapter has presented the four OMT steps used to build the three analysis models for an AFIT information system. A problem statement was developed by reading software support documentation, and consulting with present AFITSIS users and maintainers. An object model was constructed from the problem statement and problem domain. Dynamic and functional models were constructed from

system scenarios. Object model updating continued as data was viewed from different perspectives, or changes to object relationships were identified. As noted in the section on *Mapping Ternary Associations* the two approaches that were proposed to aid in the implementation of ternary and link attribute modeling concepts in object-oriented database systems still result in the loss of ternary modeling information. The proposed approaches called for expanding the ODMG-93 standard to make provisions for both of the modeling concepts freeing developers and designers from 're-inventing' code to manage such data. The object database management system standard, ODMG-93, specifically prohibits *n*-ary relationships and ignores the existence of link attributes. Therefore, in order to map the modeling of such associations into the ODBMS defined by the ODMG-93 standard the design presented calls for abstracting the relationship into an object and enforcing ternary rules through the operations provided by the relationship object.

The next chapter discusses the implementation issues which must be resolved in order to implement the AFITSIS revision. Emphasis is on the data manipulation language required to implement object models in both relational database management systems and object-oriented database management systems. The three database languages compared are the relational SQL, the SQL-like OQL, and a 'future' OQL.

IV. Implementation Issues

Introduction. A prototype of the student tracking and registration system was not implemented on the ODMG-93 ODBMS standards because such systems are still under development. This paper analyzes and designs an information system to the proposed standards to study how object-oriented analysis and design fit with the anticipated OODBMS interfaces. An advertised advantage of object-oriented database management systems is the ability to create different views of persistent data. This chapter presents views from the student tracking and registration system, and the database language (SQL and OQL) pseudo code necessary to implement those views. The first view considered is that of a student's class schedule. The second example expands upon the student class schedule object model by adding a relationship to obtain more information from the same objects. Additional models are contained in Appendix A.

Student Class Schedule. A student class schedule represents perhaps the most often used form from a school's database. The schedule represents the current student's schedule information with respect to the quarter enrolled, degree and program being pursued, and classes being taken. Figure 4.1 presents the general outline of a blank student class schedule. Figure 4.2 is the class schedule object model constructed to support the class schedule in Figure 4.1. For the sake of simplicity the object model presented lists only those attributes associated with the student class schedule.

Student Name
Box Number

Year Quarter:. Dates Inclusive

COURSE TITLE	GRD HRS	TYP	DAYS	START TIME	END TIME	BLDG	ROOM	INSTRUCTOR

SCHOOL: DEGREE:

CLASS: PROGRAM:

Figure 4.1 Student class schedule

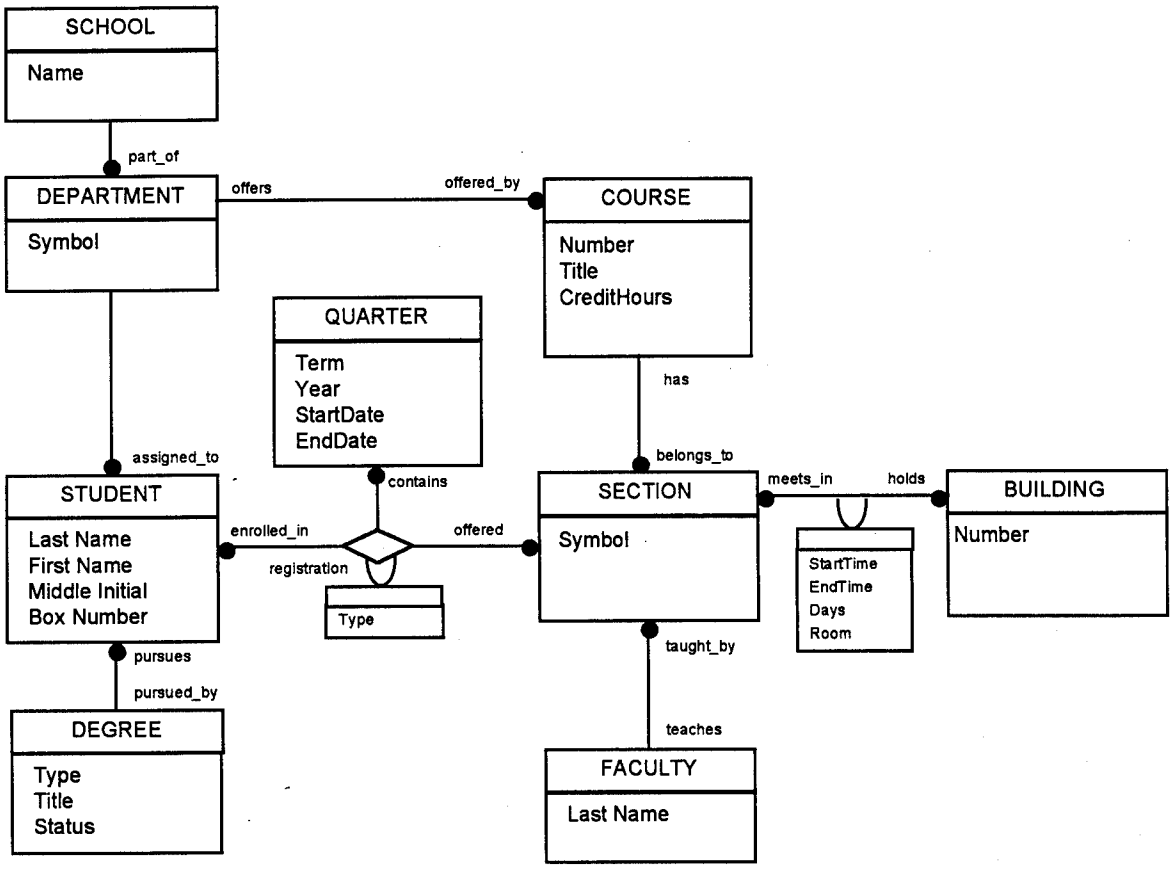


Figure 4.2 Student class schedule object model

As described in Chapter III the object model shown in Figure 4.2 can be designed for either a relational or object database management system. The examples that follow assume that the relational tables and objects have been created and databases are populated with valid data. In the SQL and OQL pseudo-code presented the bold faced text represent reserved words. As part of the analysis and design it is necessary to decide on what information is required from the database and to construct the algorithms necessary to get that data. The following algorithm constructs a student class schedule from the object diagram; the algorithm assumes input of the student's social security account number and the quarter of interest (asterisks in the left margin identify the portion of the algorithm discussed in the next subsection):

Build schedule on student's identification and quarter.
Obtain Instance of Student and Quarter.
** Output Student's First Name, Middle Initial, Last Name, and Box Number.
Output Year, Term, Start Date and End Date for instance of quarter.
** (With student id and quarter) Traverse Student-Section-Quarter enrolled_in ternary association to get instances of sections student is enrolled in.
** For each section instance:
** Output Section Symbol.
** Output enrolled_in ternary association linked attribute Grade Type.
** Traverse Section-Course belongs_to association to get instance of Course.
** Output Course's Number, Title, and Credit Hours.
** For each course instance:
** Traverse Course-Department offers association to obtain instance of Department.
** Output Department symbol.
** Traverse Section-Faculty taught_by association to obtain instance of Faculty.
** Output Faculty's Last Name.
** Traverse Section-Room meets_in association to obtain instance of Room.
** Output Building Number.
** Output meets_in linked attributes Room, Days, Start Time, and End Time.
Traverse Student-Degree pursues association to obtain instance(s) of Degree.
Output Degree Title (Degree) and Type (Class).
Traverse Student-Department assigned_to association to obtain instance of Department.
Output assigned_to linked attribute Program.
Traverse Department-School part_of aggregation to obtain instance of School.
Output School.

SQL versus OQL Comparison. The discussion of SQL versus OQL considers the retrieval of a student's name and course information, shown by

asterisks along the left margin of the algorithm above. Examples of the full SQL and full OQL implementations of the complete algorithm creating a student schedule is documented in Appendix A along with additional views of the information system. For both implementations the user inputs are assumed to be *InputSSAN* and *InputTerm*. Assume the following relational tables exist (fields not associated with this example have been left out for simplicity):

Course_Taught_By(Course_Taught00Term_Code, Course_Prefix_Code, Course_Number,
Course_Section, Faculty_SSAN)
Grade_History(Course_Prefix_Code, Course_Number, Course_Section, Schedule00Term_Code,
SSAN, Credit_Hours, Grade_Type_Code)
Schedule(Course_Prefix_Code, Course_Number, Course_Section, Course_Start_Time,
Course_End_Time, Course_Title, Day_Code, Schedule00Building_Code,
Schedule00Room_Code, Schedule00Term_Code)
Person(SSAN, First_Name, Last_Name, Middle_Initial)
Resident_Student(SSAN, Classification_Code, AFIT_Degree_Code, Program_Code, Box_Number,
Selected_Type_Code)
Terms(Term_Code, Term)
Term_Date(Term_Code, Term_Start_Date, Term_End_Date)

From the *Person* and *Resident_Student* relational tables a view is created, *View_Person_Resident*, to simplify the amount of data being handled by the system. The *View_Person_Resident* table is then joined with tables *Terms* and *Term_Date* so a selection on the input variables *InputSSAN* and *InputTerm* will result in the student's name, box number, and term information. Next, the *Grade_History* and *Schedule* tables are joined and a set of courses extracted on *InputSSAN* and *InputTerm*. The courses of this set are those in which the student is enrolled for the given term. Joining the *Person* and *Course_Taught_By* tables and using the courses' identifiers, *InputSSAN*, and *InputTerm* the instructor for each course may be obtained. The pseudo-code (with SQL) is shown below:

create view View_Person_Resident(SSAN, Last_Name, First_Name, Middle_Initial,
Box_Number) **as**
(**select** SSAN, Last_Name, First_Name, Middle_Initial

```
        from Person, Resident_Student
        where Person.SSAN = Resident_Student.SSAN )

select First_Name||' '||Middle_Initial||' '||Last_Name, Box_Number, Term||' : '||Term_Start_Date||'
    to ' ||Term_End_Date
into Name, Box_Number, Term_Info
        from View_Person_Resident, Terms, Term_Date
        where View_Person_Resident.SSAN = InputSSAN
        and Terms.Term_Code = InputTerm
        and Term_Date.Term_Code = InputTerm

Output( Name, Box_Number, Term_Info )

select Grade_History.Course_Prefix_Code, Grade_History.Course_Number,
    Grade_History.Course_Section, Course_Title, Grade_History.Credit_Hours,
    Grade_Type_Code, NVL(Day_Code,'TBA'), Course_Start_Time, Course_End_Time,
    NVL(Schedule00Building_Code, 'TBA'), NVL(Schedule00Room_Code, 'TBA')
into Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    Grading_Type, Days, Start_Time, End_Time, Building, Room
        from Grade_History, Schedule
        where Grade_History.SSAN = InputSSAN
        and Grade_History.Schedule00Term_Code = InputTerm
        and Schedule.Schedule00Term_Code = InputTerm
        and Grade_History.Course_Prefix_Code = Schedule.Course_Prefix_Code
        and Grade_History.Course_Number = Schedule.Course_Number
        and Grade_History.Course_Section = Schedule.Course_Section
Output( Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    Grading_Type, Days, Start_Time, End_Time, Building, Room )

LocalCourse_Prefix_Code = Course_Prefix_Number
LocalCourse_Number = Course_Number
LocalCourse_Section = Course_Section

select Last_Name
into Instructor
        from Person, Course_Taught_By
        where Person.SSAN = Course_Taught_By.Faculty_SSAN
        and Course_Taught_By.Course_Taught00Term_Code = InputTerm
        and Course_Taught_By.Course_Prefix_Code = LocalCourse_Prefix_Code
        and Course_Taught_By.Course_Number = LocalCourse_Number
        and Course_Taught_By.Course_Section = LocalCourse_Section
Output( Instructor )
```

In a similar fashion queries were constructed on objects of an object-oriented database system. As with the relational example above it is assumed that the objects from Figure 4.2 have been created, data is available, and the user inputs are *InputSSAN* and

InputTerm. The pseudo-code (using proposed ODMG-93 OQL) to construct the student schedule follows:

```
select distinct LocalStudent
        from x in Students
        where x.SSAN = InputSSAN

Output( LocalStudent.First_Name, LocalStudent.Middle_Initial, LocalStudent.Last_Name,
        LocalStudent. Box_Number )

select ScheduleClasses
        from x in Student.Registration
             y in Quarter.Registration
        where x.SSAN = InputSSAN
        and y.Term = InputTerm.Term
        and y.Year = InputTerm.Year

for index in first(ScheduleClasses) .. last(ScheduleClasses) loop
        element( select LocalCourse
                        from x in ScheduleClasses[index].belongs_to)
        select LocalDepartment
                from x in Course
                     y in x.offered_by
                where x = LocalCourse

        Output( LocalDepartment.Symbol, LocalCourse.Number,
                ScheduleClasses[index].Symbol, LocalCourse.Title, LocalCourse.CreditHours,
                ScheduleClasses[index].Registration.Type )

        select LocalBuilding
                from x in ScheduleClasses[index]
                     y in x.meets_in
                where x = LocalCourse

        Output( ScheduleClasses[index].Plan.Days, ScheduleClasses[index].Plan.Start_Time,
                ScheduleClasses[index].Plan.End_Time, LocalBuilding.Number,
                ScheduleClasses[index].Plan.Room )

        element( select LocalInstructor
                        from x in ScheduleClasses[index].taught_by )

        Output( LocalInstructor.Last_Name )
end loop
```

The student information is obtained by selecting an instance of a student with *InputSSAN* from the collection of students. Next, a set of sections is created,

ScheduleClasses, from the *Student-Quarter-Section* tuple by selecting on *InputSSAN* and *InputTerm*. This query on the ternary association assumes the mapping of a ternary association is provided for as discussed earlier in Chapter III. The set of sections is traversed from the first instance of section to the last. Additional traversals, nested within the outer for-loop, provide related section information, such as, the course title, number of credit hours, and the instructor.

The OQL as defined in the ODMG-93 standard is SQL-like. The difference at the programmer level is not having to deal with table operations. The programmer must be familiar with the object model to know which objects to query and relationships to traverse. The ODMG-93 standard is a way of bridging from today's database binding to a more robust, programming language-like future binding. This future binding will require a complex implementation allowing programmers to use the same pointer for either transient or persistent class instances and a sophisticated query language interpreter to locate, bind, and execute the methods for query functions. A pseudo-code example of this future binding to construct the student schedule queries for a student's name and course information is presented below:

```
LocalStudent      : Student          := students[ SSAN = InputSSAN ]

Output( LocalStudent.First_Name, LocalStudent.Middle_Initial, LocalStudent.Last_Name,
        LocalStudent. Box_Number )

ScheduledClasses                     : set("Section")
ScheduledClasses := sections{ class_roster Student[ SSAN = InputSSAN ]
                    and offered_in Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ] }

LocalCourse       : Course
LocalDepartment   : Department
LocalBuilding     : Building
LocalInstructor   : Faculty

for index in first(ScheduleClasses) .. last(ScheduleClasses) loop
        LocalCourse       := courses[ has Section[ ScheduledClasses[index] ] ]
        LocalDepartment   := departments[ offer Course[ LocalCourse ] ]
        LocalBuilding     := buildings[hold Section[ ScheduledClasses[index] ] ]
```

```
        LocalInstructor          := faculty[teaches Section[ScheduledClasses[index] ] ]
        LINK_ATTRIBUTE := TERNARY_RELATIONSHIP[ Student[ SSAN = InputSSAN ],
                                Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ],
                                Section[ ScheduledClasses[index] ]
        LINK_ATTRIBUTES := LINK_RELATIONSHIP[ Section[ ScheduledClasses[index] ],
                                Building[ LocalBuilding ] ]

        Output( LocalDepartment.Symbol, LocalCourse.Number,
                ScheduledClasses[index].Symbol, LocalCourse.Title, LocalCourse.CreditHours,
                LINK_ATTRIBUTE.Type, LINK_ATTRIBUTES.Days,
                LINK_ATTRIBUTES.Start_Time, LINK_ATTRIBUTES.End_Time,
                LocalBuilding.Number, LINK_ATTRIBUTES.Room,
                LocalInstructor.Last_Name )
end loop
```

In the future binding example, the *LocalStudent* is declared a student type and assigned the value of the instance of a student from the collection of students whose SSAN is equal to the *InputSSAN*. Relevant student information can be obtained. In the next query, a collection of sections, *ScheduleClasses*, is created for the student by forming a predicate from *InputSSAN* and *InputTerm*. The *ScheduleClasses* collection is then traversed much like an array in a programming language. This handling of objects in collections allows the programmer to work in one language and not have to perform queries in a database language and functions in a programming language. The benefit is that programmers can learn one language, and effectively use these 'future OQL' bindings much like functions within the programming language. These functions must be provided by the DBMS and would be used like I/O programming language operations.

Class Roster. The class roster depicted in Figures 4.3 and 4.4 repeat much of the same information and many of the same objects as examined in Figures 4.1 and 4.2, respectively. Two of the differences encountered in the class roster example are the addition of the graduates many-to-one *Student-Quarter* relationship with link attribute Symbol and the additional attributes for Student and Degree, such as Program_Sequence_Code and Program_Code, respectively. Much of the querying and

SCHOOL:
Year Quarter: Dates Inclusive

COURSE	TITLE	HRS	INSTRUCTOR	DAYS	START TIME	END TIME	BLDG	ROOM

STUDENTS (LAST NAME, FIRST M.I.)	SSAN	TYPE	PROGRAM	GRADUATION CODE

Figure 4.3 Class roster

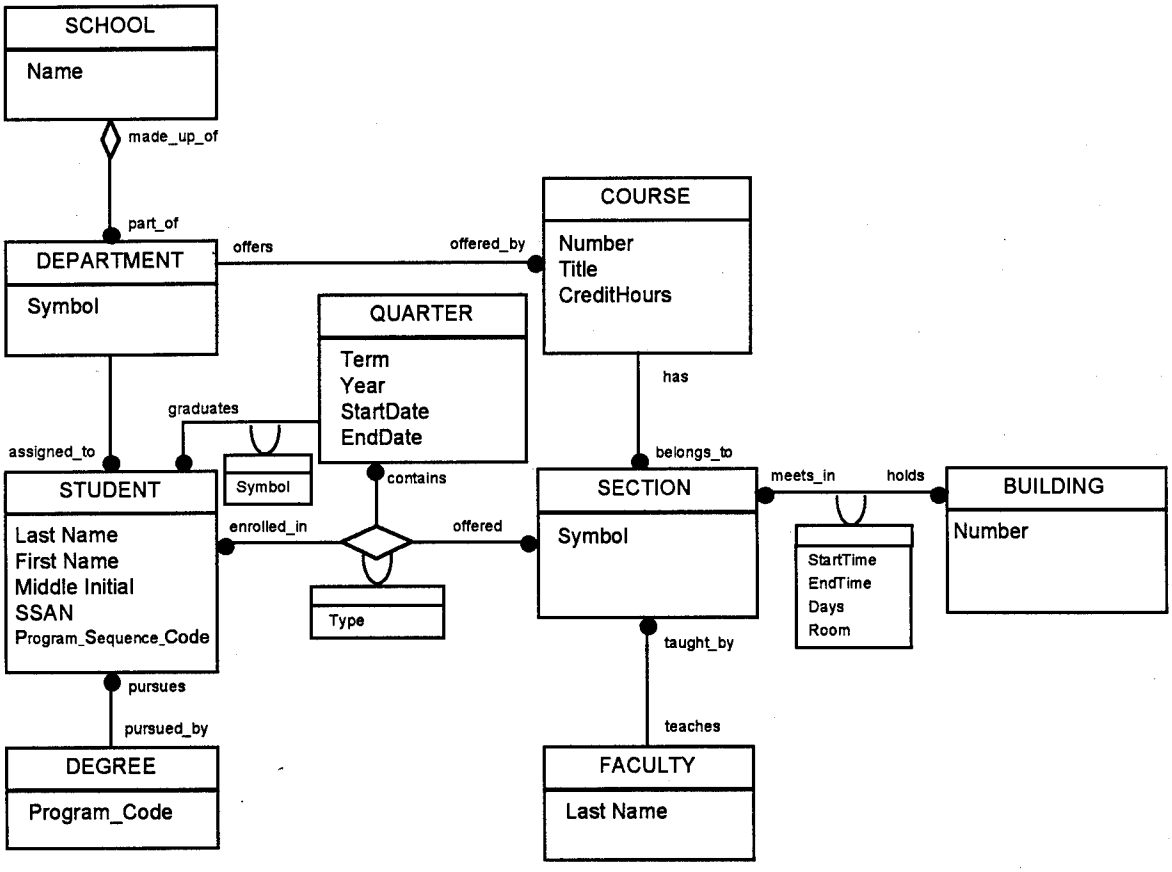


Figure 4.4 Class Roster Object Model

data manipulation will resemble that already performed in the earlier student class schedule example, and is provided in more detail in Appendix A.

The following excerpt from the class roster algorithm demonstrates the use of the new attributes and graduates association with link attributes (assume a user input of course identifier and quarter of interest):

```
For each student instance:
        Output Last Name, First Name, Middle Initial, SSAN, and Program_Sequence_Code.
        Output Student-Section-Quarter ternary association link attribute Type.
        Traverse Student-Degree pursues association to obtain instance of Degree.
        Output Program_Code.
        Traverse Student-Quarter graduates association to obtain instance of Quarter.
        Output Year and link attribute Symbol.
```

SQL versus OQL. The class roster comparison traverses the new *Student-Quarter* graduates relationship and outputs the Program_Code link attribute. In the SQL pseudo-code example provided the *Program Sequences* table and earlier *View_Person_Resident* table are joined and the student's Program_Sequence_Code selected from the set of students whose Program_Code, Class_Code, and Year_Prefix are equivalent between tables and the student's SSAN matches the one entered by the user. The pseudo-code is shown below for the addition of the graduates link attribute association. Assume that all previous tables are available and the *Program_Sequence* table has also already been created.

```
Program_Sequences( Program_Code, Class_Code, Year_Prefix, Program_Sequence_Code )


    select Program_Sequence_Code
        from Program_Sequences, View_Person_Resident
        where View_Person_Resident.SSAN = LocalSSAN
        and Program_Sequences.Program_Code = View_Person_Resident.Program_Code
        and Program_Sequences.Class_Code = View_Person_Resident.Class_Code
        and Program_Sequences.Year_Prefix = View_Person_Resident.Year_Prefix
Output( Program_Sequence_Code )
```

The SQL example shows that without embedded pointers to define relationships at the implementation level a table is required with keys allowing for a relationship among the objects to be modeled and to store the link attribute data. In much the same way the object oriented database (as expanded for link attributes) allows the user to select the relationship and output the link attribute. The pseudo-code (with proposed OQL) to construct class roster is:

```
select LINKATTRIBUTE
        from x in Students
             y in x.graduates
        where x = StudentRoster[index]
Output( LocalDegree.Program_Code, LocalQuarter.Year, LINKATTRIBUTE.Symbol)
```

In contrast to these SQL and OQL approaches, the future OQL approach will need to provide the functions necessary to select a given object from a collection identified through the relationship given a predicate identifying a single instance of the related object or an instance of that object. The pseudo-code (with proposed 'future OQL') to construct class roster is:

```
LocalDegree     : Degree := degree[ pursued_by Student[ StudentRoster[ index] ]
LocalQuarter    : Quarter := quarter[ graduates Student[ StudentRoster[ index] ]
LINK_ATTR       := LINK_RELATIONSHIP[ Section[ ScheduledClasses[ index]],
                                      Building[ LocalBuilding ] ]

Output( LocalDegree.Program_Code, LocalQuarter.Year, LINK_ATTR.Symbol )
```

Both OQL examples demonstrate that much like relational database implementations, when a relationship between two or more objects is required, that relationship must be defined by a pointer structure between the two objects.

Summary. This chapter has presented the pseudo-code implementation for the construction of a student's class schedule and a department's (instructor's) class rosters assuming the user provides the necessary input variable(s) from which to construct a predicate. The implementation is presented for both a relational database and an object-

oriented database. By comparing implementations we can see that the fundamental requirement of DBMSs being designed to the ODMG-93 standard is to provide the programmer with the functionality necessary to make the database and the programming languages unified.

V. Summary and Conclusions

Summary. The most important result of this research is the AFITSIS object modeling technique detailed analysis and design. The analysis resulted in the documentation of general student information system requirements, multiple object models developed to capture the data and attributes of the objects involved in such an information system, a dynamic model, resulting state transition diagrams, capturing the control aspects of the system, and the functional model, resulting in data flow diagrams, capturing the functional aspects of the system. The design resulted in the presentation of the mapping of objects and associations contained in an object model DBMS.

As part of the analysis it was shown that the ODMG-93 ODBMS standard does not plan for direct implementation of *n*-ary relationships or link attribute modeling concepts. To meet the requirement of implementing an *n*-ary relationship a modeling approach was presented in which the *n*-ary relationship is abstracted as an object. This relationship object will then contain all the associations and inverse associations with the *n* objects forming the n-ary relationship. As it was pointed out in Chapter III, this same modeling approach will allow the modeling of link attributes on any relationship.

This research has compared and contrasted the two different implementation approaches for mapping a database design from an object model to its respective database and database language, relational and object databases and SQL and OQL, respectively. The biggest advantage is the elimination of the intermediate table model required for the relational database implementation and allowing the implementation to map directly from the object model to the database language.

In terms of having the ability of meeting information system requirements and comparing the two database implementations, there seems to be no impact in adopting an object-oriented database versus a relational database. As discussed in Chapter I, Joseph points to five principles for moving to the next generation database:

1. Lack of expressive data modeling. Although the implementation of the system with relational tables prohibits direct modeling of complex data types the developers and maintainers are able to work around this by storing information in tables and extracting information by joining tables with related information and selecting on predicates.

2. Impedance mismatch does not exist because the AFITSIS application is built, maintained, and operates in the SQL/Oracle environment to which it is specifically targeted. A mismatch could occur if for some unlikely reason the database was required to operate outside this host environment.

3. Interactive performance does not support next generation applications. The AFITSIS implementation does inherit the expense of the relational query, but with the speed of the hardware and by developers/maintainers optimizing queries to only select and pass necessary fields, the additional expense may be minimized. From the end user perspective the operations performed are not restrictive in performance of their daily duties.

4. Mechanisms to support long transactions. These mechanisms are non-existent in AFITSIS, but are not required. Operations required of the database are of fixed length, with known input, and known output. The transactions within AFITSIS are of short-duration with little or no user interface. The user simply builds a request for information, such as a student schedule, and the system returns the request.

5. Lack of schema evolution mechanism. Although a shortfall in AFITSIS, the maintainers and administrators work around this limitation by developing a process in which database versions are archived. For example, when students graduate from AFIT

they are no longer active, so their student information is removed from the Student Tracking and Registration System database and archived in an 'alumni' file. Developing this process allows the system users to treat all data in a single state -- the current state.

Conclusions. The three research objectives, as stated in Chapter I, were:

1. Research current AFITSIS requirements, design, and implementation. Document current system performance benchmarks and maintainability metrics for comparison with prototype system.

2. Use object-oriented methods to design and implement an object-oriented database management system (OODBMS) prototype system. Use current requirements as stated by AFIT/SC and STARS users (performance and user interface requirements). The design structure will be created with possible upgrades in mind.

3. Test OODBMS prototype against current requirements. Compare performance benchmarks and maintainability predictions with current RDBMS.

The research was successful in about half of the original objectives. Research of the current AFITSIS operations and software support documentation led to the creation of an application problem statement. Three analysis models, object, dynamic, and functional models were constructed to capture the requirements of the information system. Finally, from these analysis models a proof of concept detailed design was constructed to show how the information contained within the object model may be retrieved. This was presented for both a relational database, which the current AFITSIS is built upon, and a object database showing possible implementations on a proposed ODBMS standard.

Currently no ODBMSs support this ODBMS standard and the advertised unification of database and programming languages. This along with time constraints prevented a working prototype from being implemented with which to compare similar functionality from the current AFITSIS. This could be a possible area in which future research could be conducted.

There are four main conclusions that may be reached from this research:

1. Using the OMT a proposed model has been developed with which a student tracking and registration system application could be developed.

2. Examples have shown that an implementation of the information system application is possible with either a relational or object-oriented database management system.

3. Examples demonstrate that removal of the table modeling level of design allows for more straight-forward implementation of database applications; two-levels of mapping versus three. This should lead to design documentation matching implementation more closely which will aid in both development and maintenance of database applications.

4. Future object database binding will go a long way in solving the "impedance mismatch" principle of implementing database applications.

Future Plans for AFITSIS. AFIT/SC has recently acquired the hardware necessary to upgrade versions of the AFITSIS database, with others to follow at a later date, and the AFIT/SCV personnel are preparing to perform the hardware migration and software version updates. This planned upgrade will allow AFITSIS to move from PL/SQL version 6.0 to version 7, Oracle*Forms version 2.3 to version 4, and Oracle*Menus version 4.1 to version 5. This upgrade effort most likely will aid in the 'look and feel' of the AFITSIS system, but software maintainability at best will remain constant. In the conversion of the existing code it is estimated that 30% of the code will need some changes in order for the information system to operate in the new environment. This modification obviously involves a substantial amount of effort and could provide an area for future research. Specifically in the area of code reuse, automated code generation, measuring performances, and the drawbacks and benefits to performing the upgrade.

The maintainability issue needs to be studied further in terms of what the Air Force currently spends and is projected to spend in life-cycle maintenance and support costs versus what it would cost to re-engineer an information system and its associated life-cycle costs. If the latter option is viable this research would be a valuable asset in beginning this effort. Suggestions for re-engineering the system are: identify a requirements group early in the process made up of current maintainers, users, and developers to define preliminary system requirements, prototype the dynamic aspects of the system allowing the users to provide feedback on success of implementation decisions, prototype report designs and user required queries, and baseline the system requirements. The OMT models provided as part of this research provide the developers a framework from which to begin requirements definition and dynamic prototyping. All three of the models should be able to be used directly with modifications made according to changing requirements and changes to dynamic operations. The object and functional models will most likely need extending to accept the additional forms and reports that were not captured as part of this research effort.

This approach to re-engineering the AFIT information system is important for three reasons: (1) the end user is involved in the application development making their recommendations a top priority, (2) as opposed to the original development effort, a majority of requirements are known allowing developers more flexibility in their implementation, and (3) object-oriented principles and DBMS should provide better maintainable software.

Malcolm Atkinson estimated as much as 30% of a program's lines of code could be eliminated by having a single unified system in which programming and database languages are transparent [3:33]. When ODBMSs based on the ODMG-93 standard finally make it to market, an interesting case study would be to compare the effort required to generate an identical application based upon a RDBMS versus an ODBMS.

An argument for the use of object-oriented analysis, design, and programming is the ability to reuse objects. Corporate America and the DoD have been trying to justify and estimate the savings of reuse and object-oriented approaches. Despite all this attention, not a single study exists on the development of an application using two approaches -- one with reuse and the other without.

APPENDIX A

AFITSIS DESIGN

Appendix A contains the OMT object, dynamic, and functional models necessary to implement a student tracking and registration system. The object model consists of object diagrams for school, person, and course views. The dynamic model has an additional scenario, the related event trace, an application event flow diagram, a complete collection of state diagrams, and the resulting state transition table. The functional model consists of data flow diagrams for the information system application. Following these OMT models is a design mapping comparison for implementing an association OMT modeling concept in a RDBMS versus an ODBMS. Then examples are provided along with possible algorithms for extracting data from the object models presented earlier. Finally, data dictionary descriptions are provided capturing the attributes of the objects and the range of values permitted.

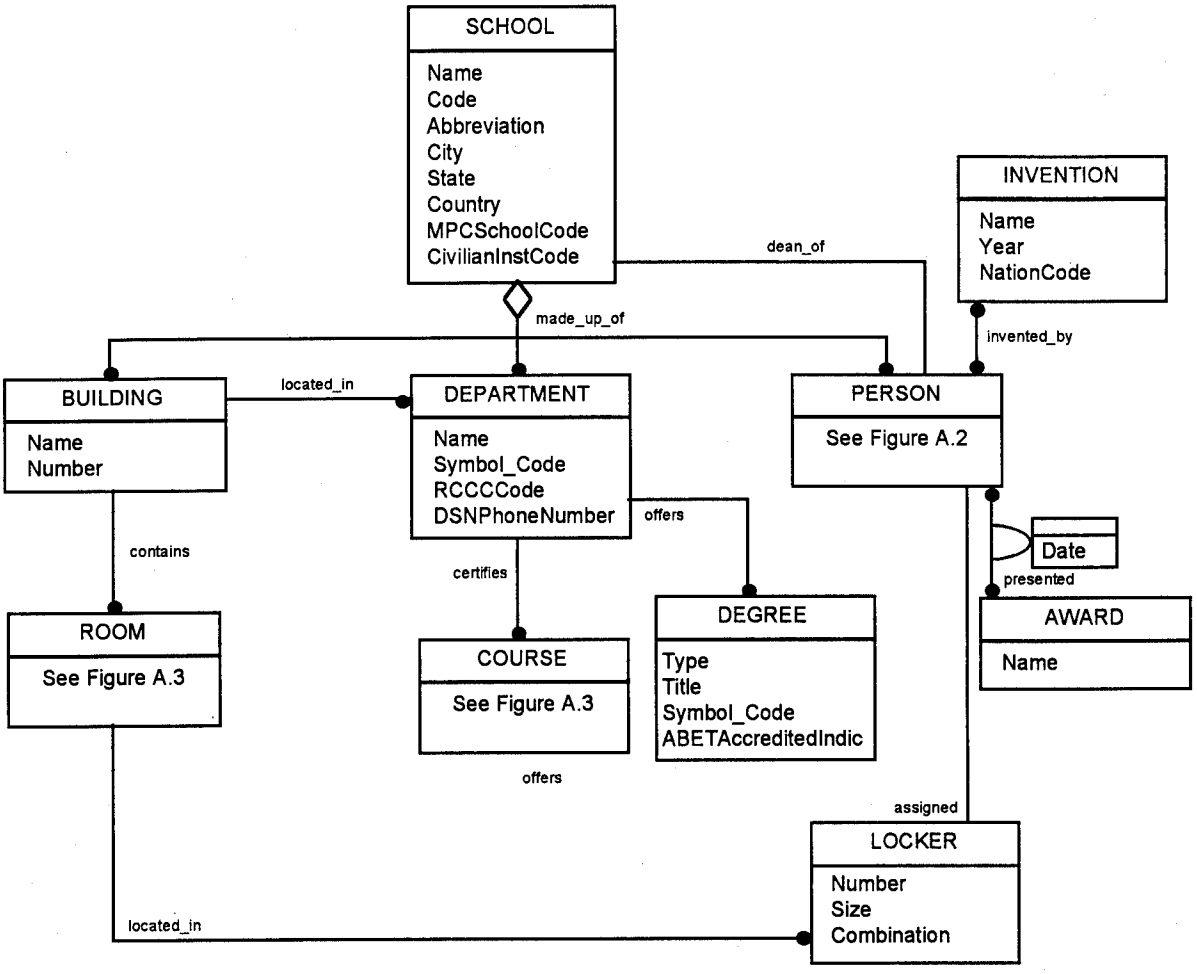


Figure A.1 School Object Diagram

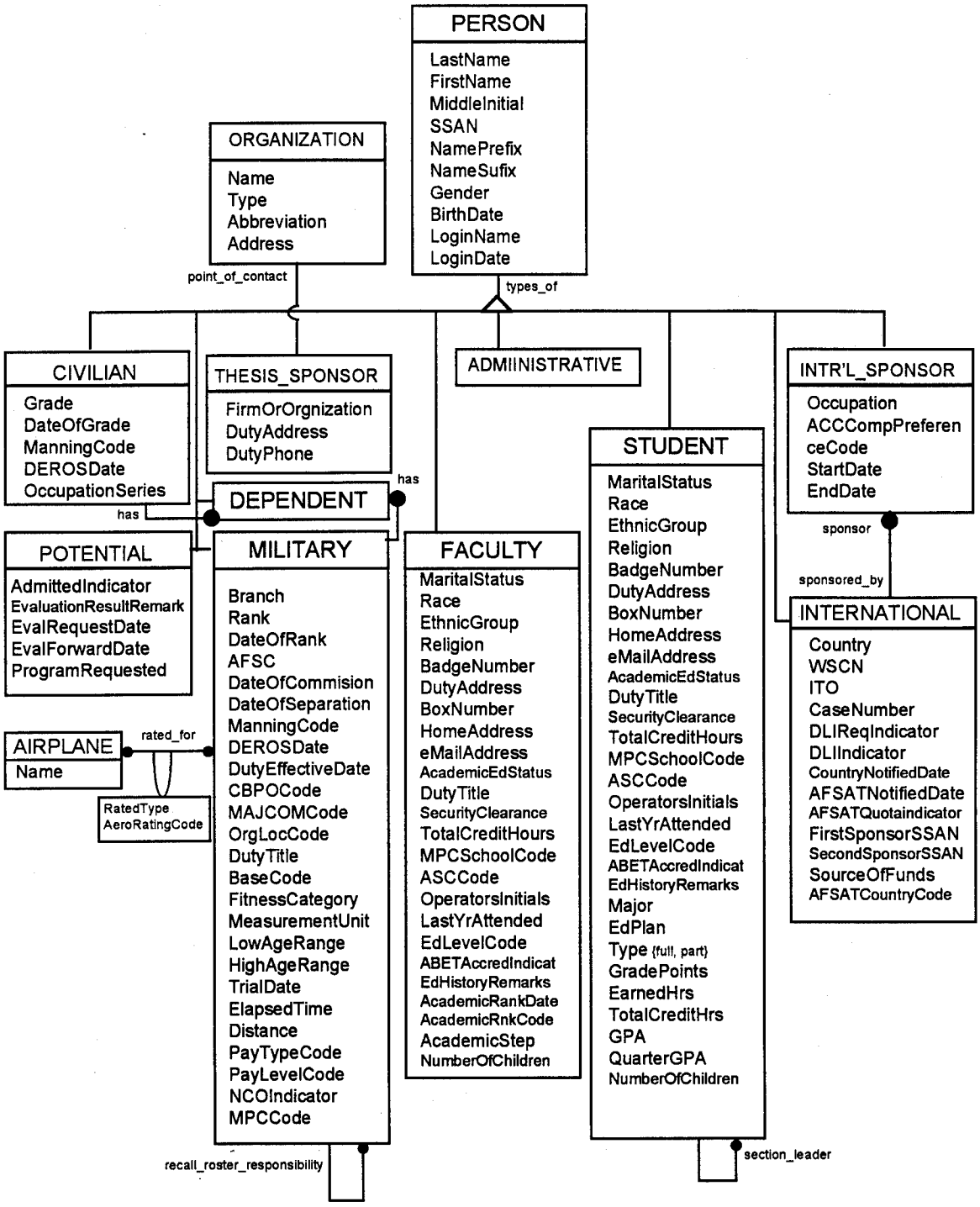


Figure A.2 Person Object Diagram

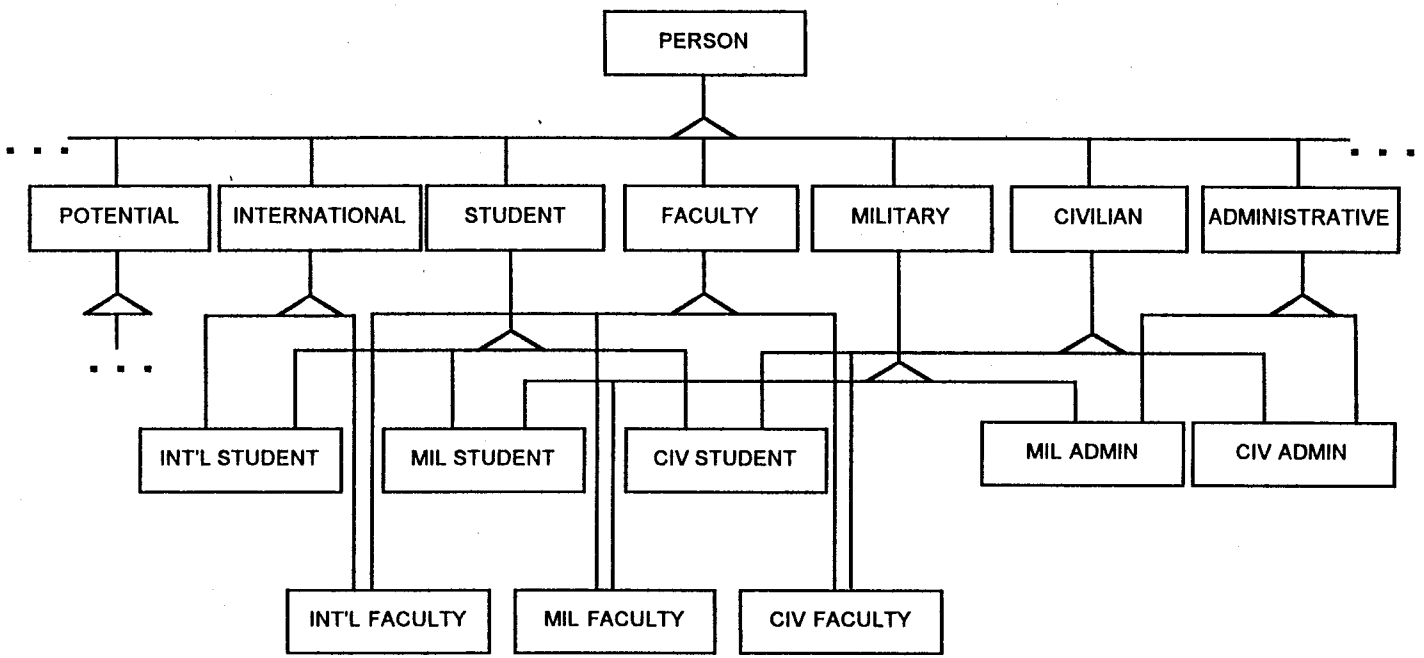


Figure A.3 Multiple Inheritance Person Object Diagram

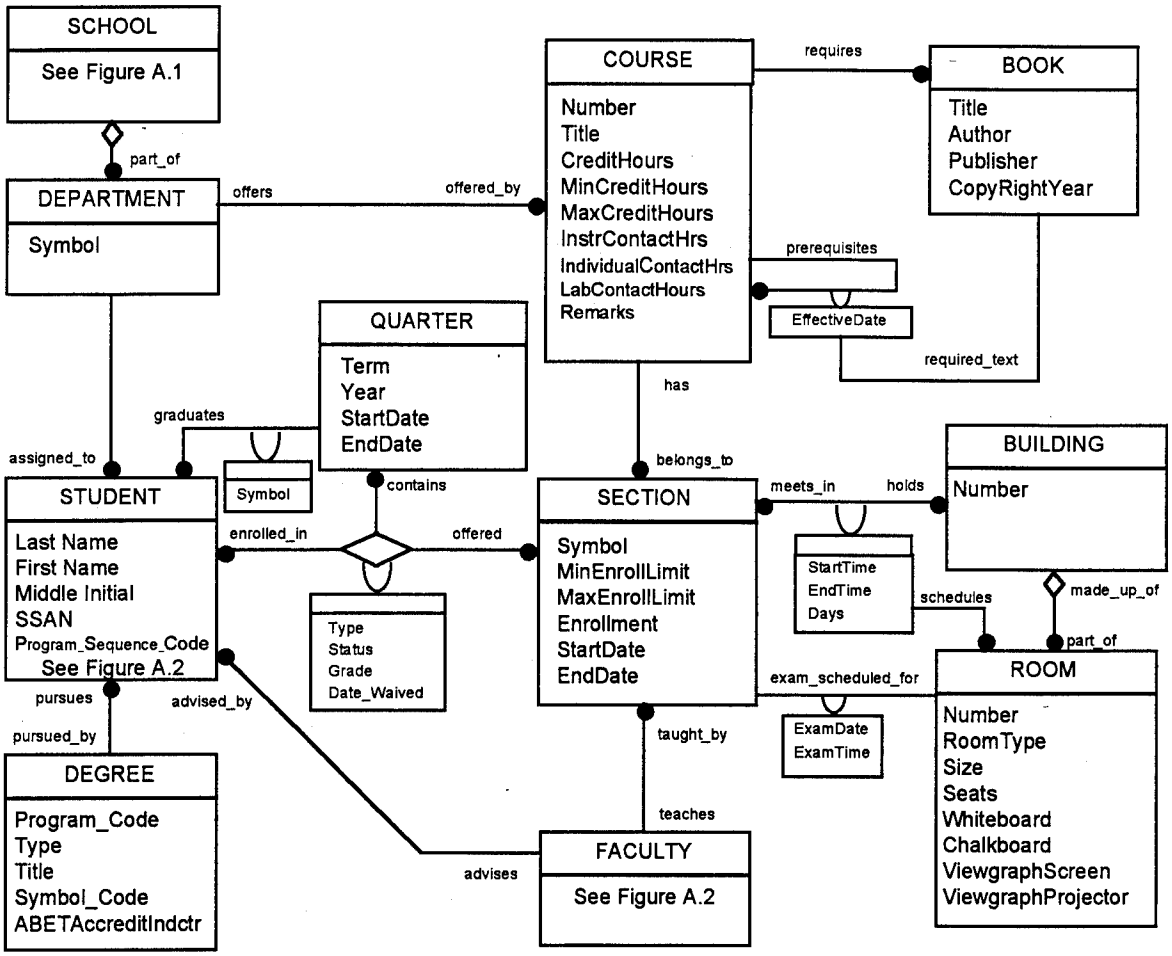


Figure A.4 Course Object Diagram

First three steps same as Figure 3.10.
The application presents student user with menu of operations (create schedule, modify schedule, view schedule, print schedule); user selects view schedule.
The application sends system view schedule request.
The system passes application the requested student schedule.
The application displays schedule and presents student user with a submenu of operations; the user selects print schedule.
The application prints schedule.
The application presents user with menu of operations; the user selects exit.
The application terminates.

Figure A.5 Student user scenario

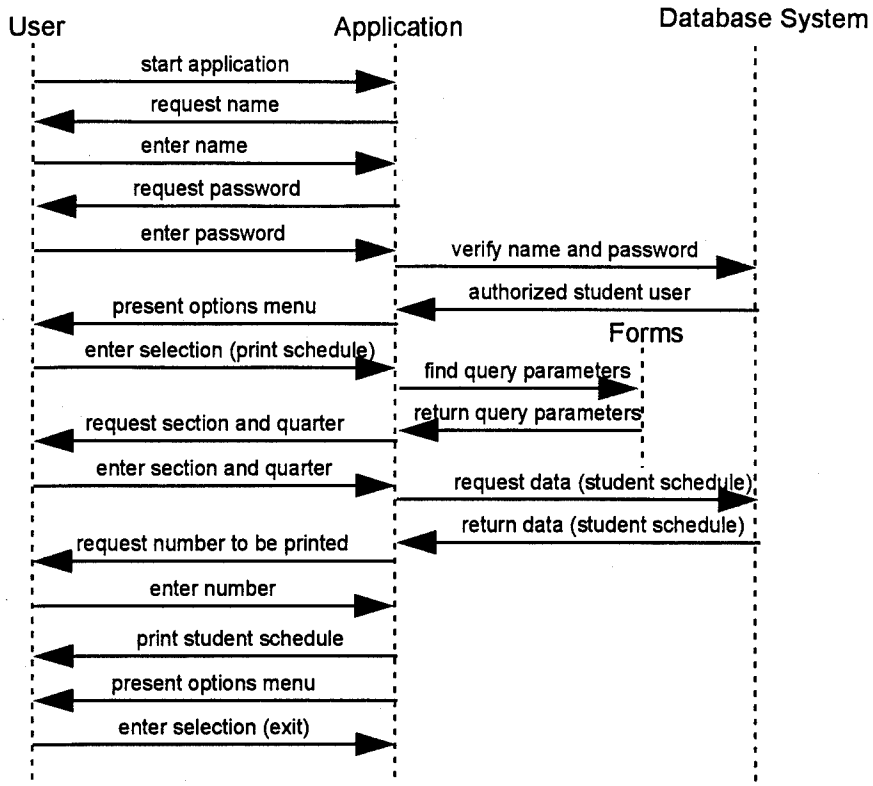


Figure A.6 Event trace for student user scenario

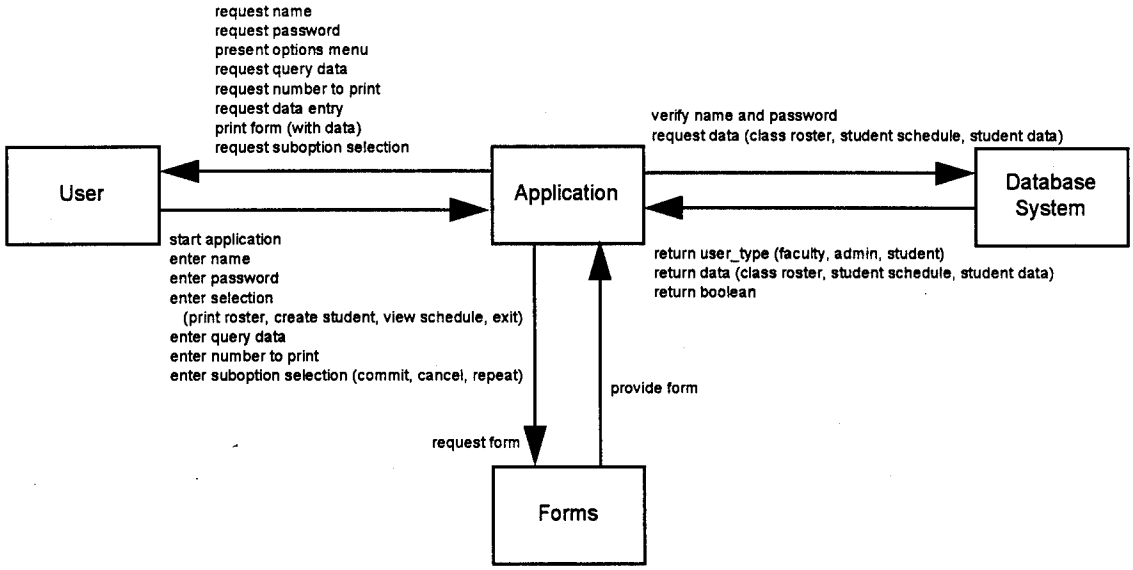


Figure A.7 Application event flow diagram

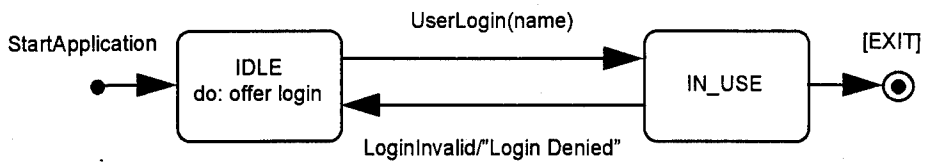


Figure A.8 Application state diagram

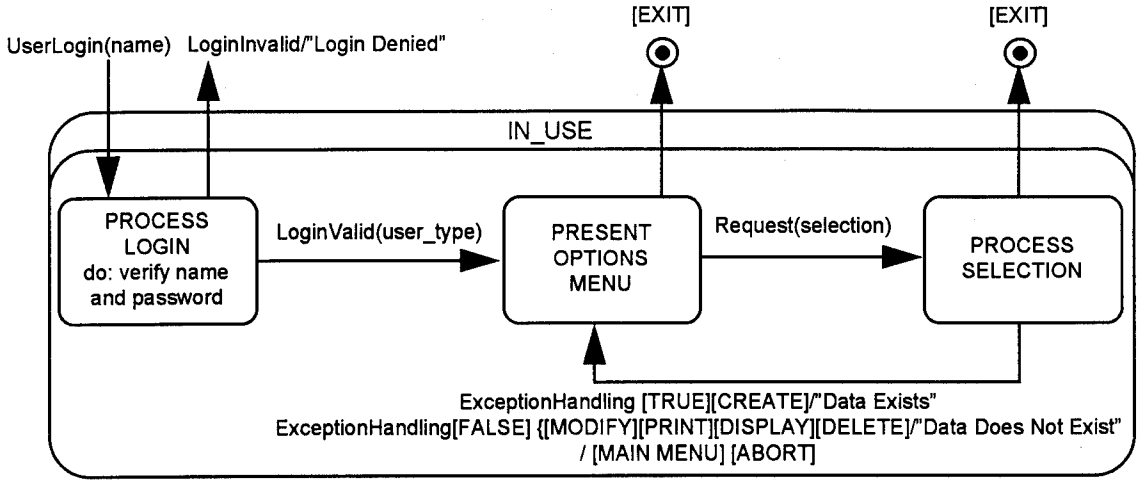


Figure A.9 Substates of In_Use state

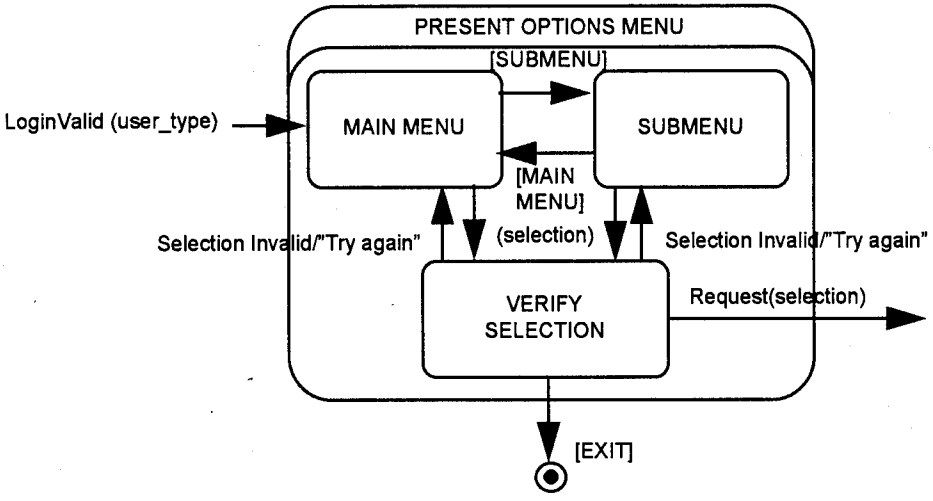


Figure A.10 Possible generalization of Present Options Menu State

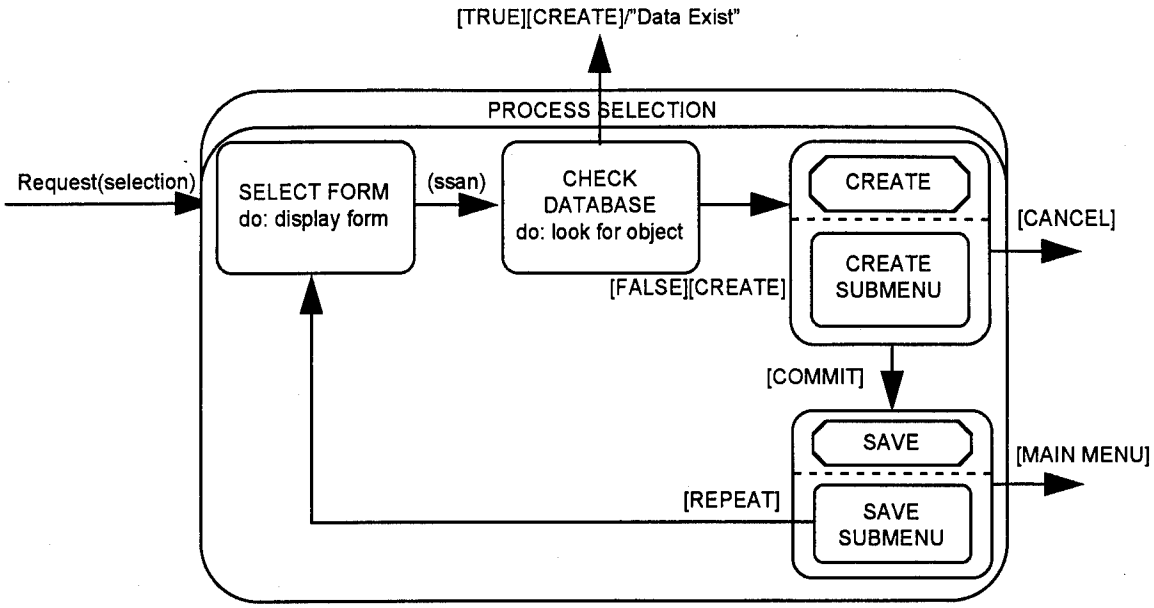


Figure A.11 Partial substates of Process Selection state

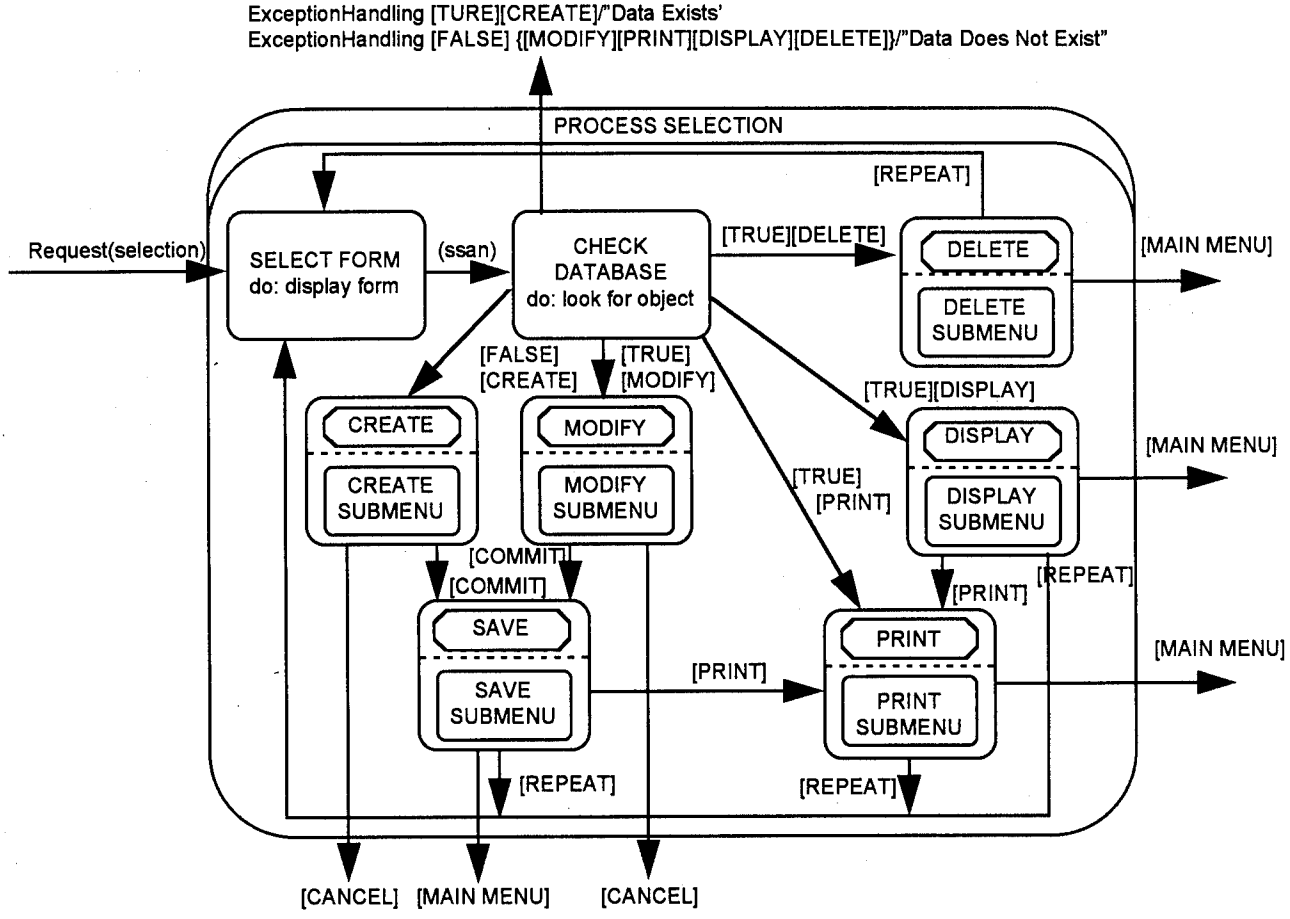


Figure A.12 Process Selection state

Current State	Event	Parameter	Guard	Next State	Action
Idle	UserLogin	name		Process Login	
Process Login	LoginValid		ADMIN	Display Admin Menu	
Process Login	LoginValid		FACULTY	Display Faculty Menu	
Process Login	LoginValid		STUDENT	Display Student Menu	
Process Login	LoginInvalid			Idle	"Login Denied"
Display Admin Menu	Request	selection		Select Form	
Display Faculty Menu	Request	selection		Select Form	
Display Student Menu	Request	selection		Select Form	
Display Student Menu			EXIT	terminal	
Select Form		SSAN		Check Database	
Check Database			TRUE, CREATE	Display Option Menu	"Data Exists"
Check Database			FALSE, CREATE	Create/Submenu	
Check Database			FALSE, MODIFY	Display Option Menu	"No Data Exists"
Check Database			FALSE, PRINT	Display Option Menu	"No Data Exists"
Check Database			FALSE, DISPLAY	Display Option Menu	"No Data Exists"
Check Database			FALSE, DELETE	Display Option Menu	"No Data Exists"
Check Database			TRUE, MODIFY	Modify/Submenu	
Check Database			TRUE, PRINT	Print/Submenu	
Check Database			TRUE, DISPLAY	Display/Submenu	
Check Database			TRUE, DELETE	Delete/Submenu	
Create/Submenu	SelectSubmenu	suboption	COMMIT	Save Data/Submenu	
Create/Submenu	SelectSubmenu	suboption	CANCEL	Display Option Menu	
Create/Submenu	SelectSubmenu	suboption	SubPRINT	Print/Submenu	
Save Data/Submenu	SelectSubmenu	suboption	REPEAT	Select Form	
Save Data/Submenu	SelectSubmenu	suboption	MAIN MENU	Display Option Menu	
Modify/Submenu	SelectSubmenu	suboption	COMMIT	SaveData/Submenu	
Modify/Submenu	SelectSubmenu	suboption	CANCEL	Display Option Menu	
Modify/Submenu	SelectSubmenu	suboption	SubPRINT	Print/Submenu	
Print/Submenu	SelectSubmenu	suboption	REPEAT	Select Form	
Print/Submenu	SelectSubmenu	suboption	MAIN MENU	Display Option Menu	
Display/Submenu	SelectSubmenu	suboption	REPEAT	Select Form	
Display/Submenu	SelectSubmenu	suboption	MAIN MENU	Display Option Menu	
Display/Submenu	SelectSubmenu	suboption	SubPRINT	Print/Submenu	
Delete/Submenu	SelectSubmenu	suboption	REPEAT	Select Form	
Delete/Submenu	SelectSubmenu	suboption	MAIN MENU	Display Option Menu	

Table A.1 State transition table for Application

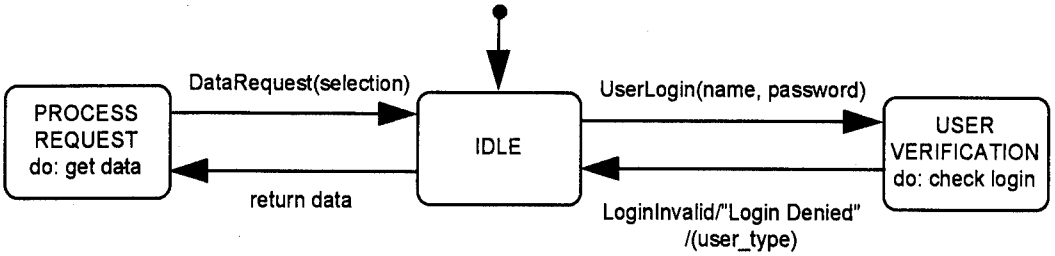


Figure A.13 State diagram for Database System

Table A.2 State transition table for Database System

Current State	Event	Parameters	Guard	Next State	Action
Idle Idle	UserLogin DataRequest	name, password form_type		User Verification Process Request	
User Verification		user_type		Idle	
Process Request		{data}		Idle	

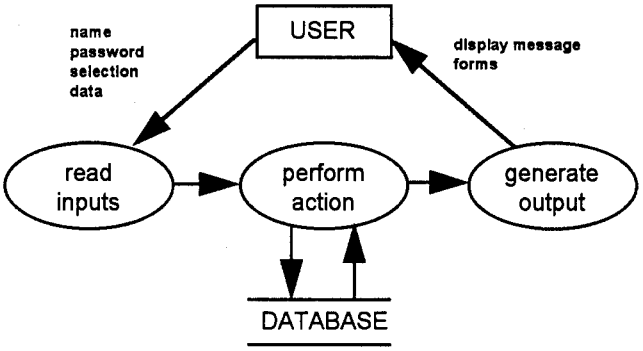


Figure A.14 STARS Application Level 0 DFD

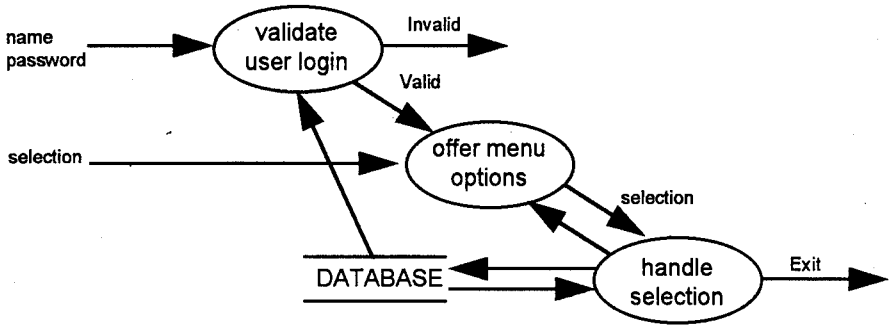


Figure A.15 Perform Action Level 1 DFD

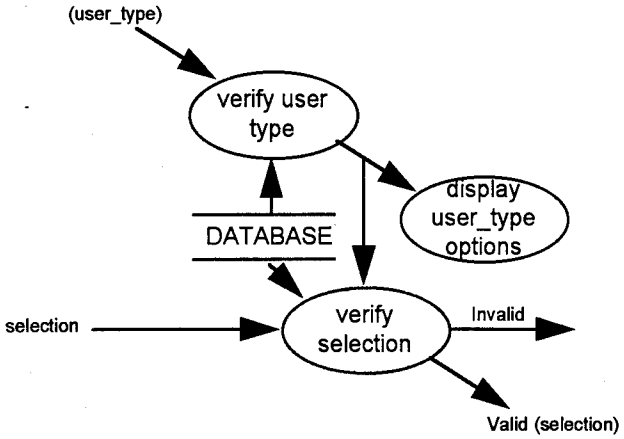


Figure A.16 Offer Menu Options Level 2 DFD

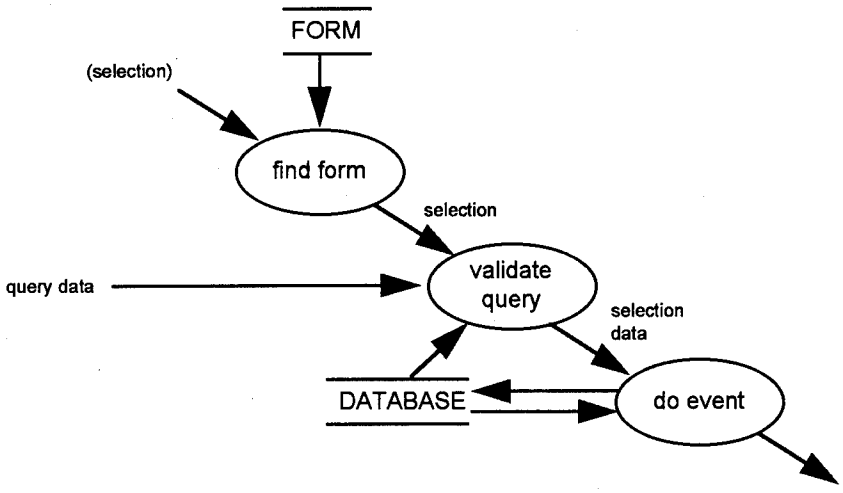


Figure A.17 Handle Selection Level 2 DFD

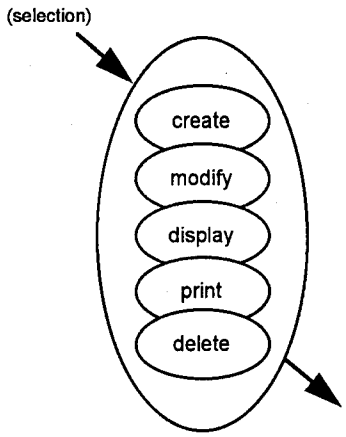


Figure A.18 Do Event Level 3 DFD

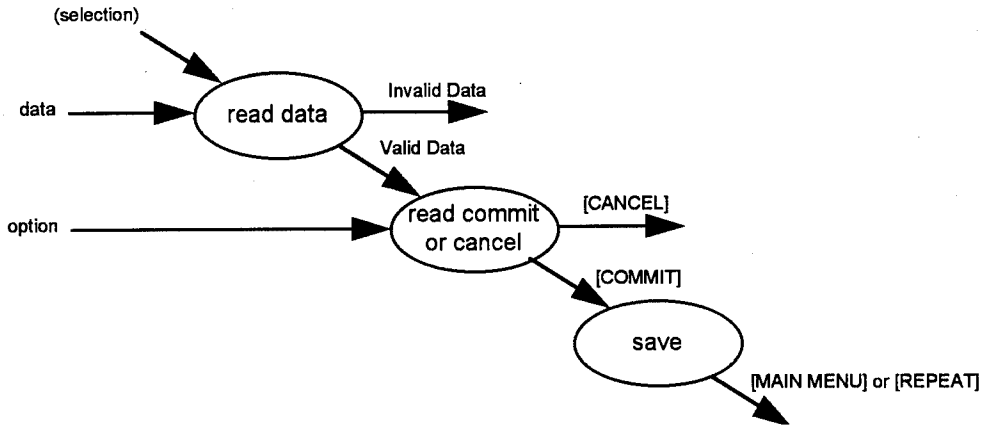


Figure A.19 Create Level 4 DFD

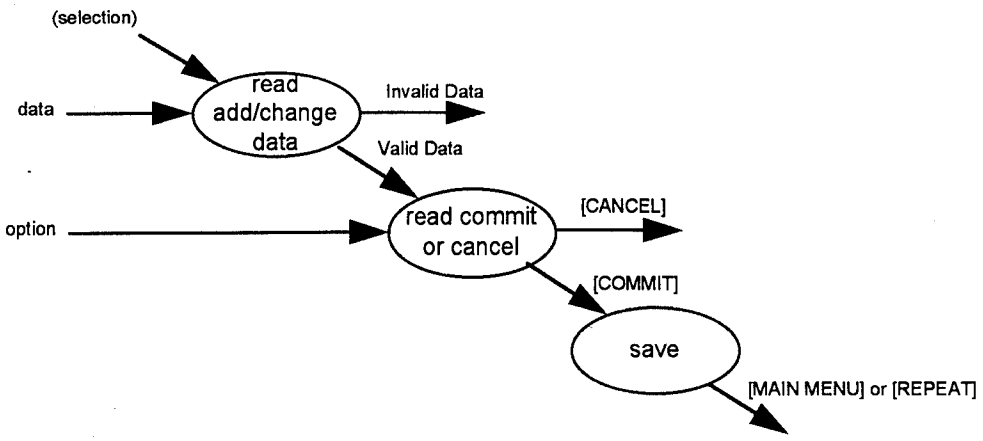


Figure A.20 Modify Level 4 DFD

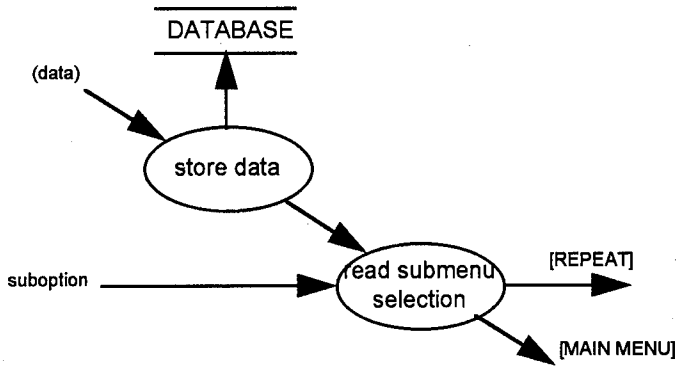


Figure A.21 Save Level 5 DFD

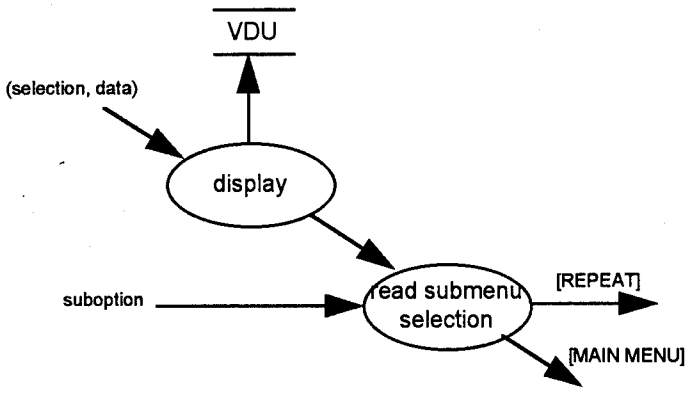


Figure A.22 Display Level 4 DFD

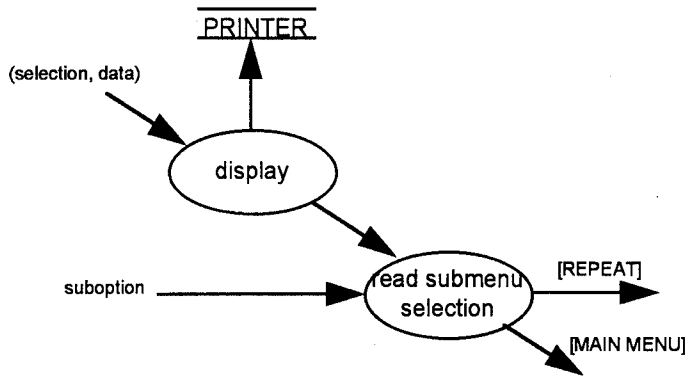


Figure A.23 Print Level 4 DFD

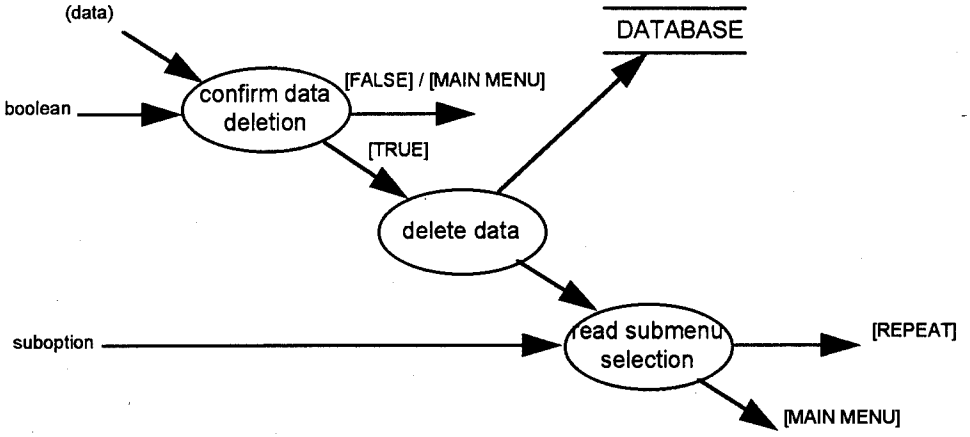
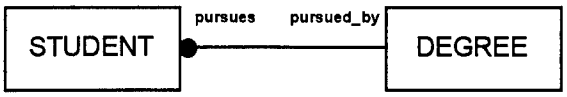


Figure A.24 Delete Level 4 DFD

Object Model



Create Person Table (Figure 3.28)
Create Student and Degree Tables, similar to Figure 3.28

Table Model

pursue Table

Attribute name	Nulls	Domain
person-ID	N	ID
degree-ID	N	ID

Candidate Key: (person-ID)
Primary Key: (person-ID)
Frequently accessed: (person-ID) (degree-ID)

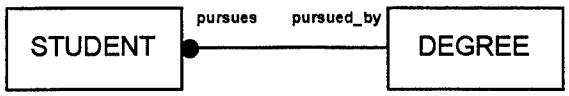
Create table and indexes for Person (Figure 3.28)
Create table and indexes for Degree (similar to Figure 3.28)

SQL Code

```
CREATE TABLE pursues
    ( person-ID     ID          not null,
      degree-ID     ID          not null,
    PRIMARY KEY (person-ID, degree-ID)
    FOREIGN KEY (degree-ID) REFERENCES Degree,
    FOREIGN KEY (person-ID) REFERENCES Person );
```

Figure A.25 Mapping association to relational database

Object Model



OQL

```
INTERFACE Student
(       EXTENT students
        KEYS SSAN, last_name )
{
        ATTRIBUTE String last_name;
        ATTRIBUTE String first_name;
        ATTRIBUTE Character middle_initial;
        ATTRIBUTE Integer(9) SSAN;
        ATTRIBUTE Integer(4) Box_Number;
        RELATIONSHIP Degree pursues INVERSE Degree::pursued_by }
```

Note: The 1:N pursues relationship will have an effect on Degree.pursued_by.

Figure A.26 Mapping association to object-oriented database

Student Class Schedule Report:

Student Name
Box Number

Year Quarter: Dates Inclusive

COURSE TITLE	GRD HRS	TYP	DAYS	START TIME	END TIME	BLDG	ROOM	INSTRUCTOR

SCHOOL: DEGREE:

CLASS: PROGRAM:

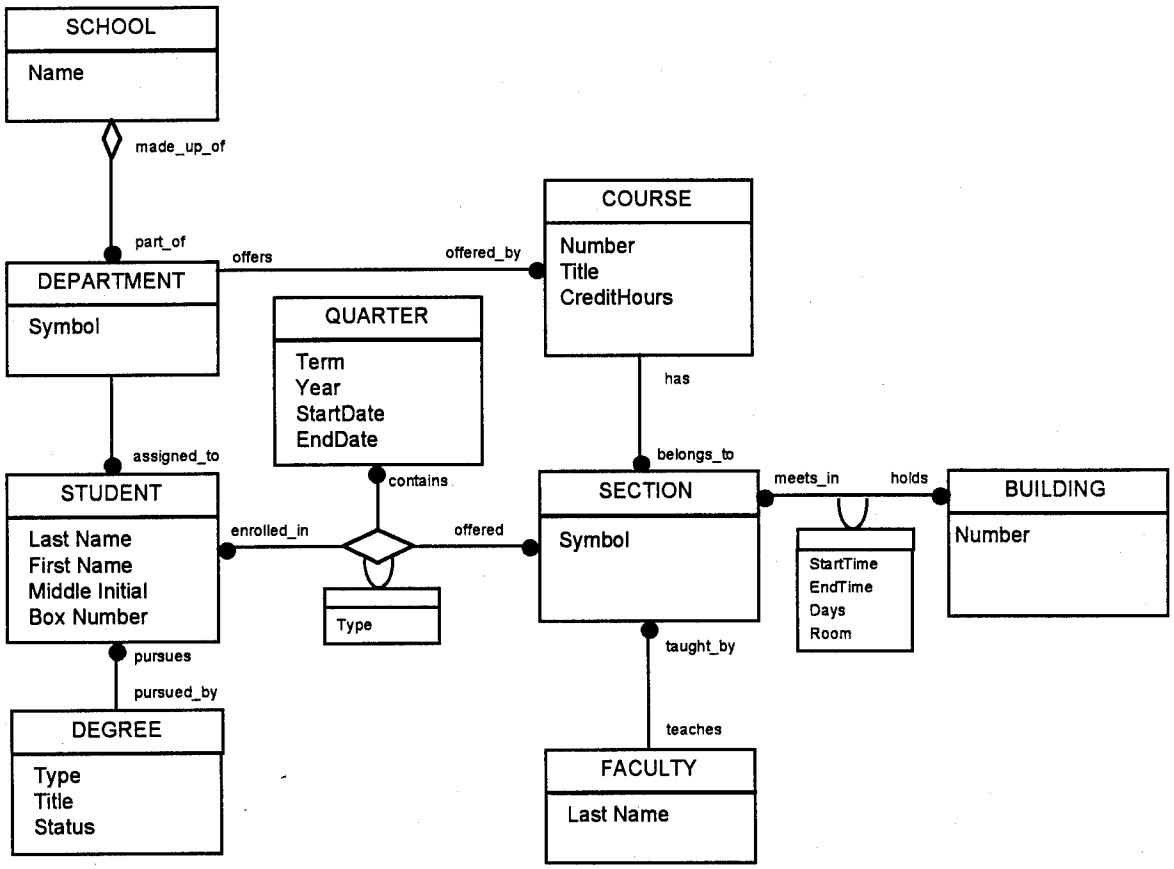


Figure A.27 Student Class Schedule Object Model

Algorithm to construct student schedule from object diagram:

Build schedule on student's identification and quarter. Input student's social security number and quarter.
Obtain Instance of Student Object and Quarter.
    Output Student's First Name, Middle Initial, Last Name, and Box Number.
    Output Year, Term, Start Date and End Date for instance of quarter.
    (With student id and quarter) Traverse Student-Section-Quarter enrolled_in ternary association
        to get instances of sections student is enrolled in.
    For each section instance:
        Output Section Symbol.
        Output enrolled_in ternary association linked attribute Grade Type.
        Traverse Section-Course belongs_to association to get instance of Course.
        Output Course's Number, Title, and Credit Hours.
        For each course instance:
            Traverse Course-Department offers association to obtain instance of Department.
            Output Department symbol.
        Traverse Section-Faculty taught_by association to obtain instance of Faculty.
        Output Faculty's Last Name.
        Traverse Section-Room meets_in association to obtain instance of Room.
        Output Building Number.
        Output meets_in linked attributes Room, Days, Start Time, and End Time.
    Traverse Student-Degree pursues association to obtain instance(s) of Degree.
    Output Degree Title (Degree), Status (Class) and Type (Program).
    Traverse Student-Department assigned_to association to obtain instance of Department.
    Traverse Department-School part_of aggregation to obtain instance of School.
    Output School.

SQL versus OQL Comparison

Assume inputs from user are *InputSSAN* and *InputTerm*.
Assume following relational tables exist:

Course_Taught_By(Course_Taught00Term_Code, Course_Prefix_Code, Course_Number,
    Course_Section, Faculty_SSAN)
Grade_History(Course_Prefix_Code, Course_Number, Course_Section, Schedule00Term_Code,
    SSAN, Credit_Hours, Grade_Type_Code)
Schedule(Course_Prefix_Code, Course_Number, Course_Section, Course_Start_Time,
    Course_End_Time, Course_Title, Day_Code, Schedule00Building_Code,
    Schedule00Room_Code, Schedule00Term_Code)
Person(SSAN, First_Name, Last_Name, Middle_Initial)
Resident_Student(SSAN, Classification_Code, AFIT_Degree_Code, Program_Code, Box_Number,
    Selected_Type_Code)
Terms(Term_Code, Term)
Term_Date(Term_Code, Term_Start_Date, Term_End_Date)
AFIT_School_Valid(AFIT_School_Code, AFIT_School)
AFIT_Degree_Valid(AFIT_Degree_Code, AFIT_Degree)
Classification_Valid(Classification_Code, Classification)
Program(Program_Code, Program)

Pseudo-code (with SQL) to construct student schedule:

```
create view View_Person_Resident( SSAN, Last_Name, First_Name, Middle_Initial,
    Box_Number ) as
        (select SSAN, Last_Name, First_Name, Middle_Initial
        from Person, Resident_Student
        where Person.SSAN = Resident_Student.SSAN )

select First_Name||' '||Middle_Initial||' '||Last_Name, Box_Number, Term||' : '||Term_Start_Date||'
    to ' ||Term_End_Date
into Name, Box_Number, Term_Info
        from View_Person_Resident, Terms, Term_Date
        where View_Person_Resident.SSAN = InputSSAN
        and Terms.Term_Code = InputTerm
        and Term_Date.Term_Code = InputTerm

Output( Name, Box_Number, Term_Info )

select Grade_History.Course_Prefix_Code, Grade_History.Course_Number,
    Grade_History.Course_Section, Course_Title, Grade_History.Credit_Hours,
    Grade_Type_Code, NVL(Day_Code,'TBA'), Course_Start_Time, Course_End_Time,
    NVL(Schedule00Building_Code, 'TBA'), NVL(Schedule00Room_Code, 'TBA')
into Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    Grading_Type, Days, Start_Time, End_Time, Building, Room
        from Grade_History, Schedule
        where Grade_History.SSAN = InputSSAN
        and Grade_History.Schedule00Term_Code = InputTerm
        and Schedule.Schedule00Term_Code = InputTerm
        and Grade_History.Course_Prefix_Code = Schedule.Course_Prefix_Code
        and Grade_History.Course_Number = Schedule.Course_Number
        and Grade_History.Course_Section = Schedule.Course_Section

Output( Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    Grading_Type, Days, Start_Time, End_Time, Building, Room )

LocalCourse_Prefix_Code = Course_Prefix_Number
LocalCourse_Number = Course_Number
LocalCourse_Section = Course_Section

select Last_Name
into Instructor
        from Person, Course_Taught_By
        where Person.SSAN = Course_Taught_By.Faculty_SSAN
        and Course_Taught_By.Course_Taught00Term_Code = InputTerm
        and Course_Taught_By.Course_Prefix_Code = LocalCourse_Prefix_Code
        and Course_Taught_By.Course_Number = LocalCourse_Number
        and Course_Taught_By.Course_Section = LocalCourse_Section

Output( Instructor )

select AFIT_School
into School
        from AFIT_School_Valid, Resident_Student
```

```
        where SSAN = InputSSAN
        and AFIT_School_Valid.AFIT_School_Code = Resident_Student.Selected_Type_Code

Output( School )

select AFIT_Degree
into Degree
        from AFIT_Degree_Valid, Resident_Student
        where SSAN = InputSSAN
        and AFIT_Degree_Valid.AFIT_Degree_Code = Resident_Student.AFIT_Degree_Code

Output( Degree )

select Classification
into Class
        from Classification_Valid, Resident_Student
        where SSAN = InputSSAN
        and Classification_Valid.Classification_Code = Resident_Student.Classification_Code

Output( Class )

select Program
        from Program, Resident_Student
        where SSAN = InputSSAN
        and Program.Program_Code = Resident_Student.Program_Code

Output( Program )
```

Pseudo-code (with proposed OQL) to construct student schedule:

Assume inputs from user are *InputSSAN* and *InputTerm*.

```
select distinct LocalStudent
        from x in Students
        where x.SSAN = InputSSAN

Output( LocalStudent.First_Name, LocalStudent.Middle_Initial, LocalStudent.Last_Name,
        LocalStudent. Box_Number )

select distinct LocalTerm
        from x in Quarters
        where x.Term = InputTerm.Term
        and x.Year = InputTerm.Year

Output( LocalTerm.Term, LocalTerm.Year, LocalTerm.Start_Date, LocalTerm.End_Date )

select ScheduledClasses
        from x in Students.Registration
             y in Quarters.Registration
        where x.SSAN = InputSSAN
```

```
        and y.Term = InputTerm.Term
        and y.Year = InputTerm.Year

for index first(ScheduleClasses) to last(ScheduleClasses) loop

        element( select LocalCourse
                        from x in ScheduleClasses[index].belongs_to)

        select LINK_ATTRIBUTE
                from x in Sections
                     y in Quarters
                     z in Students
                where x = ScheduleClasses[index]
                and y.Term = InputTerm.Term
                and y.Year = InputTerm.Year
                and z.SSAN = InputSSAN

        select LocalDepartment
                from x in Course
                     y in x.offered_by
                where x = LocalCourse
        Output( LocalDepartment.Symbol, LocalCourse.Number,
                ScheduleClasses[index].Symbol, LocalCourse.Title, LocalCourse.CreditHours,
                LINK_ATTRIBUTE.Type )

        select LINK_ATTRIBUTES
                from x in Sections
                     y in x.meets_in
                where x = ScheduleClasses[index]

        select LocalBuilding
                from x in ScheduleClasses[index]
                     y in x.meets_in
                where x = LocalCourse
        Output( LINK_ATTRIBUTES.Days, LINK_ATTRIBUTES.Start_Time,
                LINK_ATTRIBUTES.End_Time, LocalBuilding.Number,
                LINK_ATTRIBUTES.Room )

        element( select LocalInstructor
                        from x in ScheduleClasses[index].taught_by )
        Output( LocalInstructor.Last_Name )

end loop

select distinct LocalDegree
        from x in Students
             y in x.pursues
        where x.SSAN = InputSSAN

select distinct LocalDepartment
        from x in Students
```

```
            y in x.assigned_to
        where x.SSAN = InputSSAN

    select distinct LocalSchool
        from x in Students
             y in x.assigned_to
             z in y.part_of
        where x.SSAN = InputSSAN

    Output( LocalSchool.School, LocalDegree.Title, LocalDegree.Status, LocalDegree.Type )
```

Pseudo-code (with proposed OQL) to construct student schedule:

Assume inputs from user are *InputSSAN* and *InputTerm*.

```
LocalStudent     : Student          := students[ SSAN = InputSSAN ]

Output( LocalStudent.First_Name, LocalStudent.Middle_Initial, LocalStudent.Last_Name,
        LocalStudent. Box_Number )

LocalTerm        : Quarter          := quarter[ Term = InputTerm.Term, Year = InputTerm.Year ]

Output( LocalTerm.Term, LocalTerm.Year, LocalTerm.Start_Date, LocalTerm.End_Date )

ScheduledClasses                    : set("Section")
ScheduledClasses := sections{ class_roster Student[ SSAN = InputSSAN ]
                    and offered_in Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ] }


LocalCourse                 : Course
LocalDepartment             : Department
LocalBuilding               : Building
LocalInstructor             : Faculty

for index in first(ScheduleClasses) .. last(ScheduleClasses) loop
        LocalCourse             := courses[ has Section[ ScheduledClasses[index] ] ]
        LocalDepartment         := departments[ offer Course[ LocalCourse ] ]
        LocalBuilding           := buildings[hold Section[ ScheduledClasses[index] ] ]
        LocalInstructor         := faculty[teaches Section[ScheduledClasses[index] ] ]
        LINK_ATTRIBUTE := TERNARY_RELATIONSHIP[ Student[ SSAN = InputSSAN ],
                                Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ],
                                Section[ ScheduledClasses[index] ]
        LINK_ATTRIBUTES := LINK_RELATIONSHIP[ Section[ ScheduledClasses[index] ],
                                Building[ LocalBuilding ] ]

        Output( LocalDepartment.Symbol, LocalCourse.Number,
                ScheduledClasses[index].Symbol, LocalCourse.Title, LocalCourse.CreditHours,
                LINK_ATTRIBUTE.Type, LINK_ATTRIBUTES.Days,
                LINK_ATTRIBUTES.Start_Time, LINK_ATTRIBUTES.End_Time,
```

```
                    LocalBuilding.Number, LINK_ATTRIBUTES.Room,
                    LocalInstructor.Last_Name )
end loop

LocalDegree := degree[ pursued_by[Student = LocalStudent] ]
LocalDepartment := department[ has[Student = LocalStudent] ]
LocalSchool := school[ made_up_of[Department = LocalDepartment] ]

Output( LocalSchool.School, LocalDegree.Title, LocalDegree.Status, LocalDegree.Type )
```

Class Roster:

SCHOOL:
Year Quarter: Dates Inclusive

COURSE	TITLE	HRS	INSTRUCTOR	DAYS	START TIME	END TIME	BLDG	ROOM

STUDENTS (LAST NAME, FIRST M.I.)	SSAN	TYPE	PROGRAM	GRADUATION CODE

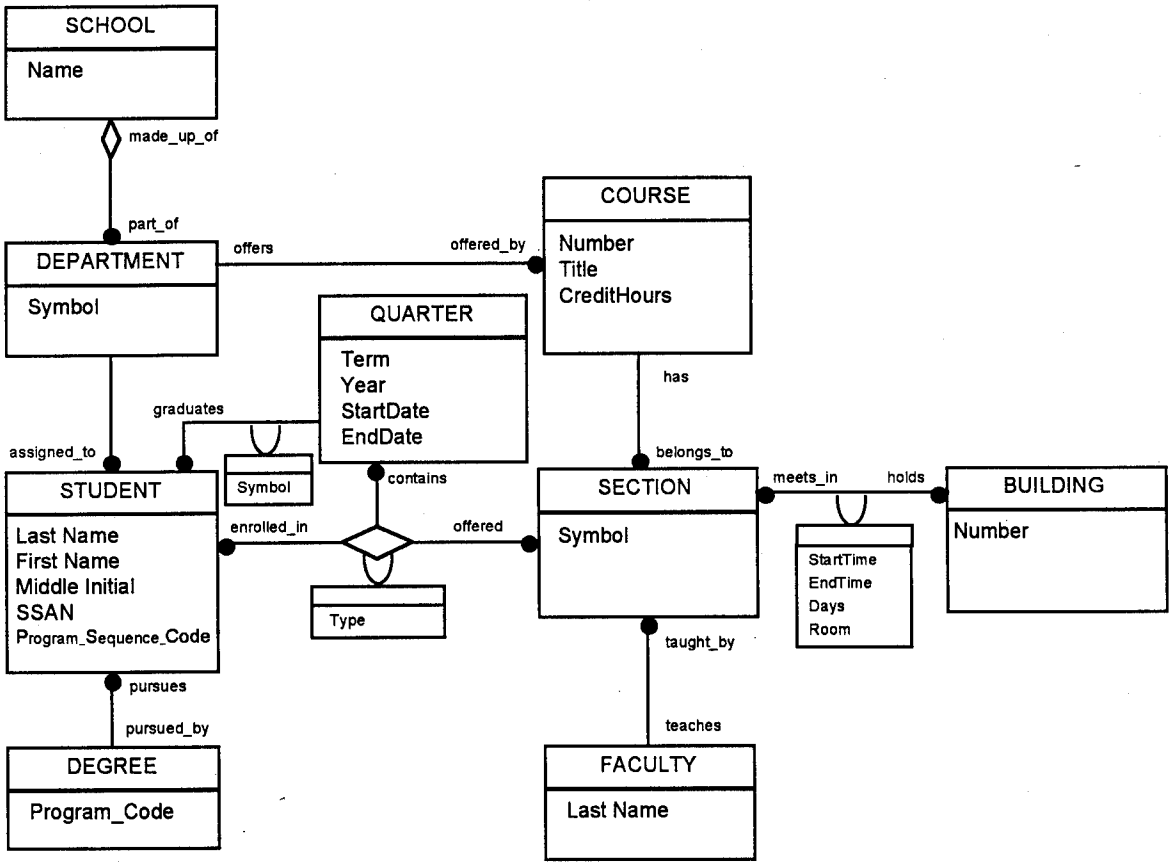


Figure A.28 Class Roster Object Model

Algorithm to construct class roster from object diagram:

Build class roster on course identification (Department Symbol, Course Number, Course Section) and Quarter. Input course identification and quarter.
Obtain Instance of Department, Course, Section, and Quarter.
    Output Department Symbol, Course Number, Section Symbol, Course Title, and CreditHours.
    Traverse Department-School part_of aggregation to obtain instance of School.
    Output School Name.
    Output Year, Term, Start Date and End Date for instance of quarter.
    Traverse Section-Faculty taught_by association to obtain instance of Faculty.
    Output Faculty's Last Name.
    Traverse Section-Building meets_in association to obtain instance of Building.
    Output Building Number.
    Output meets_in linked attributes Days, Start Time, End Time, and Room.
    (With Course Section and quarter) Traverse Student-Section-Quarter ternary association to get instances of students enrolled in section for the quarter.
    For each student instance:
        Output Last Name, First Name, Middle Initial, SSAN, and ASC_Code.
        Output Student-Section-Quarter ternary association link attribute Type.
        Traverse Student-Degree pursues association to obtain instance of Degree.
        Output Program_Code.
        Traverse Student-Quarter graduates association to obtain instance of Quarter.
        Output Year and link attribute Symbol.

SQL versus OQL Comparison

Assume inputs from user are *InputCourse_Prefix, InputCourse_Number, InputCourse_Section,* and *InputTerm*.
Assume following relational tables exist:

Course_Taught_By(Course_Taught00Term_Code, Course_Prefix_Code, Course_Number, Course_Section, Faculty_SSAN)
Grade_History(Course_Prefix_Code, Course_Number, Course_Section, Schedule00Term_Code, SSAN, Credit_Hours, Grade_Type_Code)
Schedule(Course_Prefix_Code, Course_Number, Course_Section, Course_Start_Time, Course_End_Time, Course_Title, Day_Code, Schedule00Building_Code, Schedule00Room_Code, Schedule00Term_Code, AFIT_School_Code)
Person(SSAN, First_Name, Last_Name, Middle_Initial)
Resident_Student(SSAN, Program_Code, Class_Code, Program_Year_Prefix)
Terms(Term_Code, Term)
Term_Date(Term_Code, Term_Start_Date, Term_End_Date)
AFIT_School_Valid(AFIT_School_Code, AFIT_School)
Program_Sequences(Program_Code, Class_Code, Year_Prefix, Program_Sequence_Code)

Pseudo-code (with SQL) to construct class roster:

```
select AFIT_School
into School
        from AFIT_School_Valid, Schedule
        where AFIT_School_Valid.AFIT_School_Code = Schedule.AFIT_School_Code
Output( School )
```

```
select SSAN, Course_Title, Grade_History.Credit_Hours, Grade_Type_Code,
    NVL(Day_Code,'TBA'), Course_Start_Time, Course_End_Time,
    NVL(Schedule00Building_Code, 'TBA'), NVL(Schedule00Room_Code, 'TBA')
into Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    Grading_Type, Days, Start_Time, End_Time, Building, Room
        from Grade_History, Schedule
        where Grade_History.Schedule00Term_Code = InputTerm
        and Schedule.Schedule00Term_Code = InputTerm
        and Grade_History.Course_Prefix_Code = InputCourse_Prefix
        and Grade_History.Course_Number = InputCourse_Number
        and Grade_History.Course_Section = InputCourse_Section
        and Schedule.Course_Prefix_Code = Grade_History.Course_Prefix_Code
        and Schedule.Course_Number = Grade_History.Course_Number
        and Schedule.Course_Section = Grade_History.Course_Section
Output( Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Credit_Hours,
    SSAN, Grading_Type, Days, Start_Time, End_Time, Building, Room )

Local SSAN = Grade_History.SSAN

create view View_Person_Resident( SSAN, Last_Name, First_Name, Middle_Initial,
    Class_Code, Program_Code, Program_Year_Prefix ) as
        (select SSAN, Last_Name, First_Name, Middle_Initial, Class_Code, Program_Code,
        Program_Year_Prefix
        from Person, Resident_Student
        where Person.SSAN = Resident_Student.SSAN )

select First_Name||' '||Middle_Initial||' '||Last_Name, Class_Code, Term||' : '||Term_Start_Date||' to
    ' ||Term_End_Date
into Name, Class_Code, Term_Info
        from View_Person_Resident, Terms, Term_Date
        where View_Person_Resident.SSAN = LocalSSAN
        and Terms.Term_Code = InputTerm
        and Term_Date.Term_Code = InputTerm
Output( Name, Term_Info, Class_Code )

select Last_Name
into Instructor
        from Person, Course_Taught_By
        where Person.SSAN = Course_Taught_By.Faculty_SSAN
        and Course_Taught_By.Course_Taught00Term_Code = InputTerm
        and Course_Taught_By.Course_Prefix_Code = InputCourse_Prefix
        and Course_Taught_By.Course_Number = InputLocalCourse_Number
        and Course_Taught_By.Course_Section = InputCourse_Section
Output( Instructor )

select Program_Sequence_Code
        from Program_Sequences, View_Person_Resident
        where View_Person_Resident.SSAN = LocalSSAN
        and Program_Sequences.Program_Code = View_Person_Resident.Program_Code
        and Program_Sequences.Class_Code = View_Person_Resident.Class_Code
```

```
        and Program_Sequences.Year_Prefix = View_Person_Resident.Year_Prefix
Output( Program_Sequence_Code )
```

Pseudo-code (with proposed OQL) to construct class roster:

Assume inputs from user are *InputCourse_Prefix, InputCourse_Number, InputCourse_Section,* and *InputTerm.*

```
select distinct LocalSchool
        from x in Department
                y in x.part_of
        where x.Symbol = InputCourse_Prefix
Output( LocalSchool.School )

select distinct LocalTerm
        from x in Quarters
        where x.Term = InputTerm.Term
        and x.Year = InputTerm.Year
Output( LocalTerm.Term, LocalTerm.Year, LocalTerm.Start_Date, LocalTerm.End_Date )

select distinct LocalDepartment
        from x in Departments
        where x.Symbol = InputCourse_Prefix

select distinct LocalCourse
        from x in Courses
        where x.Number = InputCourse_Number

select distinct LocalSection
        from x in Sections
        where x.Symbol = InputCourse_Section
LocalSectionOutput( LocalDepartment.Symbol, LocalCourse.Number, LocalSection.Symbol,
        LocalCourse.Title, LocalCourse.CreditHours )

element( select LocalInstructor
                from x in LocalSection.taught_by )
Output( LocalInstructor.Last_Name )

select LINKATTRIBUTES
        from x in Sections
        y in x.meets_in
        where x = InputCourse_Section
Output( LINKATTRIBUTES.Days, LINKATTRIBUTES.Start_Time,
        LINKATTRIBUTES.End_Time, LINKATTRIBUTES.Room )

select LocalBuilding
        from x in Sections
        y in x.meets_in
        where x = InputCourse_Section
Output( LocalBuilding.Number )
```

```
select StudentRoster
        from x in Sections.Registration
             y in Quarters.Registration
        where x.Symbol = InputCourse_Section
        and y.Term = InputTerm.Term
        and y.Year = InputTerm.Year

for index first(StudentRoster) to last(StudentRoster) loop

        Output( StudentRoster[index].Last_Name, StudentRoster[index].First_Name,
                StudentRoster[index].Middle_Initial, StudentRoster[index].SSAN,
                StudentRoster[index].Program_Sequence_Code )

        select LINK_ATTRIBUTE
                from x in Sections
                     y in Quarters
                     z in Students
                where x.Symbol = InputCourse_Section
                and y.Term = InputTerm.Term
                and y.Year = InputTerm.Year
                and z = StudentRoster[index]
        Output( LINKATTRIBUTE.Type )

        select LocalDegree
                from x in Students
                     y in x.pursues
                where x = StudentRoster[index]

        select LocalQuarter
                from x in Students
                     y in x.graduates
                where x = StudentRoster[index]

        select LINKATTRIBUTE
                from x in Students
                     y in x.graduates
                where x = StudentRoster[index]
        Output( LocalDegree.Program_Code, LocalQuarter.Year, LINKATTRIBUTE.Symbol)

end loop
```

Pseudo-code (with proposed OQL) to construct class roster:

Assume inputs from user are *InputCourse_Prefix, InputCourse_Number, InputCourse_Section,* and *InputTerm.*

```
LocalSchool     := school[ made_up_of Department[ Symbol = InputCourse_Prefix ] ]
Output( LocalSchool.School )

LocalTerm       : Quarter       := quarter[ Term = InputTerm.Term, Year = InputTerm.Year ]
```

```
Output( LocalTerm.Term, LocalTerm.Year, LocalTerm.Start_Date, LocalTerm.End_Date )

LocalDepartment : Department     := departments[Symbol = InputCourse_Prefix]
LocalCourse     : Course         := courses[ Number = InputCourse_Number ]
LocalSection    : Section:= sections[ Symbol = InputCourse_Section ]
Output( LocalDepartment.Symbol, LocalCourse.Number, LocalSection.Symbol,
        LocalCourse.Title, LocalCourse.CreditHours )

LocalInstructor          : Faculty:= faculty[ teaches Section[ LocalSection ]
Output( LocalInstructor.Last_Name )

LocalBuilding            := buildings[ hold Section[ LocalSection ] ]
Output( LocalBuilding.Number )

LINK_ATTRIBUTES := LINK_RELATIONSHIP[ Section[ LocalSection ],
                                      Building[ LocalBuilding ] ]

Output( LINK_ATTRIBUTES.Days, LINK_ATTRIBUTES.Start_Time,
        LINK_ATTRIBUTES.End_Time, LINK_ATTRIBUTES.Room )

StudentRoster            : set("Student")
StudentRoster := students{ enrolled_in Section[ Symbol = InputCourse_Section ]
                  and offered_in Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ] }

for index in first(StudentRoster) .. last(StudentRoster) loop

        Output( StudentRoster[index].Last_Name, StudentRoster[index].First_Name,
                StudentRoster[index].Middle_Initial, StudentRoster[index].SSAN,
                StudentRoster[index].Program_Sequence_Code )

        LINK_ATTRIBUTE := TERNARY_RELATIONSHIP[ Student[ StudentRoster[index] ],
                                    Quarter[ Term = InputTerm.Term, Year = InputTerm.Year ],
                                    Section[ LocalSection ]
        Output( LINK_ATTRIBUTE.Type )

        LocalDegree     : Degree := degree[ pursued_by Student[ StudentRoster[ index] ]
        LocalQuarter    : Quarter := quarter[ graduates Student[ StudentRoster[ index] ]
        LINK_ATTR       := LINK_RELATIONSHIP[ Section[ ScheduledClasses[ index]],
                                              Building[ LocalBuilding ] ]

        Output( LocalDegree.Program_Code, LocalQuarter.Year, LINK_ATTR.Symbol )

end loop
```

Assign Student Academic Advisor Form:

Student:

Last Name, First Name MI SSAN Rank Graduation Code

Academic Advisor:

Last Name First Name MI SSAN Rank Department

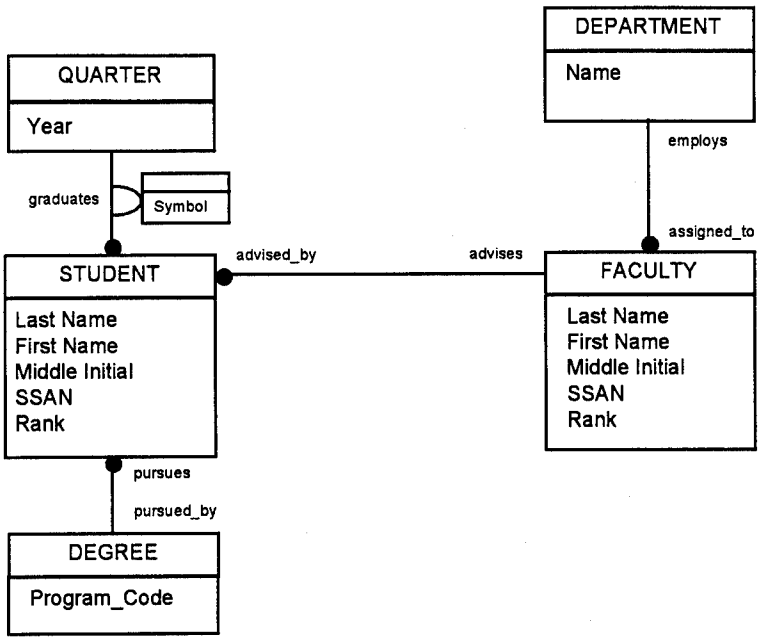


Figure A.29 Assign Student Academic Advisor Object Model

Algorithm to construct assign student academic advisor form from object diagram:

Build assign student academic advisor form on student identification. Input Student's social security account number. (Note: Similar procedures followed given instructor's identification.)
Obtain Instance of Student.
Output Student's Last Name, First Name, Middle Initial, SSAN, and Rank.
Traverse Student-Degree pursues association to obtain instance of Degree.
Output Program_Code.
Traverse Student-Quarter graduates association to obtain instance of Quarter.
Output Year and link attribute Symbol.
Traverse Student-Faculty advises association to obtain instance of Faculty
Output Faculty's Last Name, First Name, Middle Initial, SSAN, and Rank.
(With Faculty instance) Traverse Faculty-Department association for instance of Department
Output Department Name.

SQL versus OQL Comparison

Assume input from user is *InputSSAN* (Student's SSAN).
Assume following relational tables exist:

Faculty_Civilian(SSAN, Occupation_Series_Code)
Faculty_History(SSAN, Academic_Rank_Code)
Faculty_Type_Valid(Faculty_Type_Code, Faculty_Type)
Rank_History(SSAN, Grade_Rank_Abbrev)
Person(SSAN, First_Name, Last_Name, Middle_Initial, Department_Code)
Resident_Student(SSAN, Class_Code, Faculty_Advisor_SSAN)

Pseudo-code (with SQL) to construct assign advisor to student with identification.:

```
create view View_Person_Resident( SSAN, Last_Name, First_Name, Middle_Initial,
    Class_Code, Faculty_Advisor_SSAN ) as
        ( select Person.SSAN, Last_Name, First_Name, Middle_Initial, Class_Code,
        Faculty_Advisor_SSAN
        from Person, Resident_Student
        where Person.SSAN = Resident_Student.SSAN )

select First_Name||' '||Middle_Initial||' '||Last_Name, Class_Code, SSAN, Grade_Rank_Abbrev,
    Faculty_Advisor_SSAN
        into Name, Class_Code, SSAN, Rank, Advisor_SSAN
        from View_Person_Resident, Rank_History
        where View_Person_Resident.SSAN = InputSSAN
        and View_Person_Resident.SSAN = Rank_History.SSAN
Output( Name, SSAN, Rank, Class_Code )

select SSAN, Last_Name, First_Name, Middle_Initial, Department_Code,
    Academic_Rank_Code
into SSAN, Last_Name, First_Name, Middle_Initial, Department, Academic_Rank
        from Person, Faculty_History
        where Person.SSAN = Faculty_History.SSAN
        and Person.SSAN = Advisor_SSAN
Output( Last_Name, First_Name, Middle_Initial, SSAN, Department )

select Faculty_Type
        from Faculty_Type_Valid
        where Faculty_Type_Code = Academic_Rank

if Faculty_Type is Military then
        select Grade_Rank_Abbrev
        into Rank
            from Person, Rank_History
            where Person.SSAN = Rank_History.SSAN
            and Person.SSAN = Faculty_Advisor_SSAN
        Output( Rank )
else
        select Occupation_Series_Code
        into Rank
```

```
                from Person, Faculty_Civilian
                where Person.SSAN = Faculty_Civilian.SSAN
                and Person.SSAN = Faculty_Advisor_SSAN
            Output( Rank )
end if
```

Pseudo-code (with proposed OQL) to construct assign advisor to student with id:

Assume input from user is *InputSSAN* (Student's SSAN).

```
select distinct LocalStudent
        from x in Students
        where x.SSAN = InputSSAN
Output( LocalStudent.Last_Name, LocalStudent.First_Name, LocalStudent.Middle_Initial,
        LocalStudent.SSAN, LocalStudent.Rank )

select distinct LocalDegree
        from x in Students
                y in x.pursues
        where x.SSAN = InputSSAN

select distinct LocalQuarter
        from x in Students
                y in x.graduates
        where x.SSAN = InputSSAN

select ATTRIBUTE
        from x in Students
              y in x.graduates
        where x.SSAN = InputSSAN
Output( LocalDegree.Program_Code, LocalQuarter.Year, ATTRIBUTE.Symbol )

select distinct LocalFaculty
        from x in Students
                y in x.advised_by
        where x.SSAN = InputSSAN
Output( LocalFaculty.Last_Name, LocalFaculty.First_Name, LocalFaculty.Middle_Initial,
        LocalFaculty.SSAN, LocalFaculty.Rank )

select distinct LocalDepartment
        from x in Faculty
                y in x.assigned_to
        where x = LocalFaculty
Output( LocalDepartment.Name )
```

Pseudo-code (with proposed OQL) to assign academic advisor to student:

Assume input from user is *InputSSAN* (Student's SSAN).

```
LocalStudent     : Student          := student[ SSAN = InputSSAN ] ]
```

```
Output( LocalStudent.Last_Name, LocalStudent.First_Name, LocalStudent.Middle_Initial,
        LocalStudent.SSAN, LocalStudent.Rank )

LocalDegree     : Degree        := degree[ pursued_by Student[ LocalStudent ]
LocalQuarter    : Quarter       := quarter[ graduates Student[ LocalStudent ]
ATTRIBUTE       := LINK_RELATIONSHIP[ Student[ LocalStudent ],
                                      Quarter[ LocalQuarter ] ]
Output( LocalDegree.Program_Code, LocalQuarter.Year, ATTRIBUTE.Symbol )

LocalAdvisor    : Faculty       := faculty[ advises Student[ LocalStudent ] ]
Output( LocalAdvisor.Last_Name, LocalAdvisor.First_Name, LocalAdvisor.Middle_Initial,
        LocalAdvisor.SSAN, LocalAdvisor.Rank )

LocalDepartment : Department    := department[employs Faculty[ LocalAdvisor ]
Output( LocalDepartment.Name )
```

AFIT Course Catalog:

SCHOOL:
Year Quarter: Dates Inclusive

COURSE	TITLE	CREDIT HOURS	ACTIVITY TYPE	MAX ENROLL	NOTES CODE	NOTES

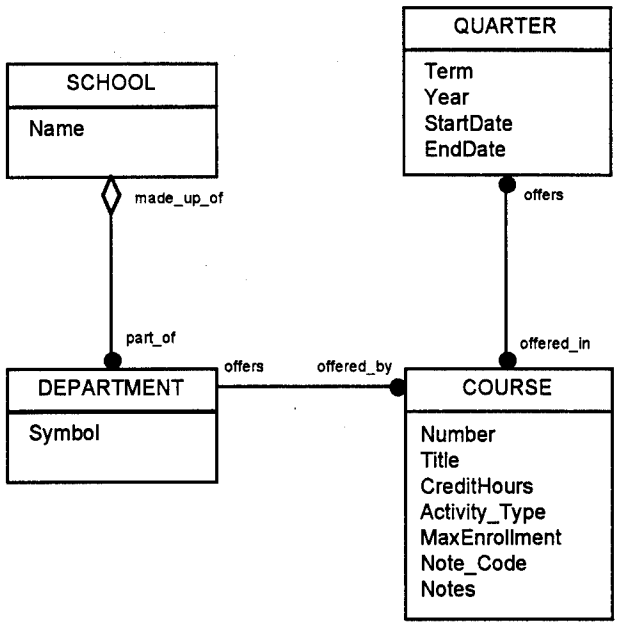


Figure A.30 AFIT Course Catalog Object Model

Student Transcript:

STUDENT NAME (LAST NAME, FIRST M.I.) SSAN PROGRAM CLASS CODE ADVISOR

QUARTER YEAR

COURSE TITLE INSTRUCTOR CREDIT HRS GRADE POINTS

•
•
•

QUARTER HOURS QUARTER GPA CUMULATIVE HOURS CUMULATIVE GPA

•
•
•

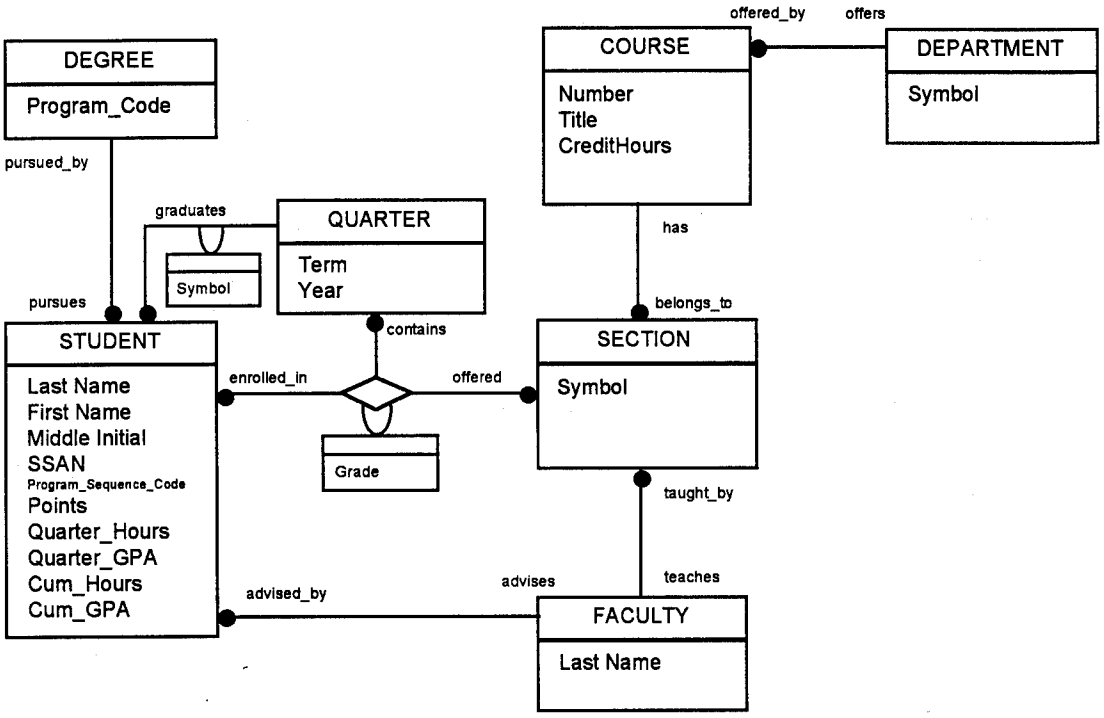


Figure A.31 Student Transcript Object Model

Person Structure Definition

Object Name: Person

Object Number:

Object Description: General model of a person

Date: 07/28/94
History: Thesis

Author: Capt Douglas Wu

Superclass: None

Components: None

Context: None

Attributes:

lastname	String	Person's last name.
firstname	String	Person's first name.
initial	character	Person's middle initial.
nameprefix	prefix type	Person's name prefix.
namesuffix	suffix type	Person's name suffix.
ssan	String	Social Security Account Number.
gender	{male, female}	
birthdate	date type	Date of birth.
loginname	String	Person's computer login name.
logindate	date type	Date of last activity.

Constraints:

Z Static Schema:
Let **SSAN_TYPE** be the set of all Social Security Account Numbers.
Let **DATE_TYPE** be the set of all possible dates.
Let **PREFIX_TYPE** be the set of all possible name prefixes.
Let **SUFFIX_TYPE** be the set of all possible name suffixes.

Person

$$
\begin{aligned}
&lastname : String \\
&firstname : String \\
&initial : Character \\
&nameprefix : PREFIX\_TYPE \\
&namesuffix : SUFFIX\_TYPE \\
&ssan : String \\
&gender : \{male, female\} \\
&birthdate : DATE\_TYPE \\
&loginname : String \\
&logindate : DATE\_TYPE
\end{aligned}
$$

Faculty Structure Definition

Object Name: Faculty

Object Number:

Object Description: General model of faculty class.

Date: 07/28/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Person

Components: None

Context: None

Attributes:

marital_status	{single, married, divorced}	
race	race type	Race.
ethnic_group	ethnic type	Ethnicity.
religion	religion type	Religion.
badge_number	badge number type	Badge number.
duty_phone	phone type	Duty telephone number.
duty_address	address type	Duty address.
home_phone	phone type	Home telephone number.
home_address	address type	Home address.
e – mail_address	String	Electronic mail address.
academic_advisees	Person set type	set of pointers to students.
thesis_advisees	Person set type	set of pointers to students.
academic_rank_date	date type	date of academic rank.
academic_rank_code	academic rank type	academic rank.
academic_step	academic step type	academic step.

Constraints:

Z Static Schema:
Let **RACE_TYPE** be the set of all race types.
Let **ETHNIC_TYPE** be the set of all ethnic types.
Let **RELIGION_TYPE** be the set of all religions.
Let **BADGE_NUMBER_TYPE** be the set of all possible badge numbers.
Let **PHONE_TYPE** be the set of all possible phone numbers.
Let **ADDRESS_TYPE** be the set of all valid addresses.

Let **PERSON_PTR_TYPE** be a pointer to a particular person.
Let **PERSON_PTR_SET_TYPE** be a set of pointers to particular people.
Let **COURSE_PTR_SET_TYPE** be a set of pointers to a quarter's courses.
Let **ACADEMIC_RANK_TYPE** be a set of valid academic ranks.
Let **ACADEMIC_STEP_TYPE** be a set of valid academic steps.

Faculty

marital_status : {*single*, *married*, *divorced*}
race : *RACE_TYPE*
ethnic_group : *ETHNIC_TYPE*
religion : *RELIGION_TYPE*
badge_number : *BADGE_NUMBER_TYPE*
duty_phone : *PHONE_TYPE*
duty_address : *ADDRESS_TYPE*
home_phone : *PHONE_TYPE*
home_address : *ADDRESS_TYPE*
e − mail_address : *String*
academic_advisees : *PERSON_PTR_SET_TYPE*
thesis_advisees : *PERSON_PTR_SET_TYPE*
academic_rank_date : *Person.DATE_TYPE*
academic_rank_code : *ACADEMIC_RANK_TYPE*
academic_step : *ACADEMIC_STEP_TYPE*

Student Structure Definition

Object Name: Student

Object Number:

Object Description: General model of a student

Date: 07/28/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Person

Components: None

Context: None

Attributes:

marital_status	{single, married, divorced}	
race	race type	Race.
ethnic_group	ethnic type	Ethnicity.
religion	religion type	Religion.
badge_number	badge number type	Badge number.
duty_phone	phone type	Duty telephone number.
duty_address	address type	Duty address.
home_phone	phone type	Home telephone number.
home_address	address type	Home address.
e − mail_address	String	Electronic mail address.
academic_advisor	Person Pointer	Pointer to academic advisor.
thesis_advisor	Person Pointer	Pointer to thesis advisor.
thesis_readers	Person set type	set of thesis readers.
ed_plan	quarter set type	set of academic quarters.
quarter_schedule	Course Pointer set	set of pointers to courses.

Constraints:

Z Static Schema:
Let **RACE_TYPE** be the set of all race types.
Let **ETHNIC_TYPE** be the set of all ethnic types.
Let **RELIGION_TYPE** be the set of all religions.
Let **BADGE_NUMBER_TYPE** be the set of all possible badge numbers.
Let **PHONE_TYPE** be the set of all possible phone numbers.
Let **ADDRESS_TYPE** be the set of all valid addresses.

Let **PERSON_PTR_TYPE** be a pointer to a particular person.
Let **PERSON_PTR_SET_TYPE** be a set of pointers to particular people.
Let **QUARTER_SET_TYPE** be a set of pointers to academic quarters.
Let **COURSE_PTR_SET_TYPE** be a set of pointers to a quarter's courses.

Student

marital_status : {*single*, *married*, *divorced*}
race : *RACE_TYPE*
ethnic_group : *ETHNIC_TYPE*
religion : *RELIGION_TYPE*
badge_number : *BADGE_NUMBER_TYPE*
duty_phone : *PHONE_TYPE*
duty_address : *ADDRESS_TYPE*
home_phone : *PHONE_TYPE*
home_address : *ADDRESS_TYPE*
e − mail_address : *String*
academic_advisor : *PERSON_PTR_TYPE*
thesis_advisor : *PERSON_PTR_TYPE*
thesis_readers : *PERSON_PTR_SET_TYPE*
ed_plan : *QUARTER_SET_TYPE*
quarter_schedule : *COURSE_PTR_SET_TYPE*

MilitaryFaculty Structure Definition

Object Name: MilitaryFaculty

Object Number:

Object Description: General model of a military faculty

Date: 07/29/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Faculty

Components: None

Context: None

Attributes:

rank	rank type	Military rank.
branch	branch type	Branch of service.
dateofrank	date type	Date of rank.
AFSC	AFSC type	AFSC code.
aeroratingcode	rating type	Aero rating.
dateofcommission	date type	Date of commission.
dateofseparation	date type	Date of separation.
manningcode		
DEROSdate	date type	

Constraints:

Z Static Schema:
Let **RANK_TYPE** be the set of military rank types.
Let **BRANCH_TYPE** be the set of all armed service branches types.
Let **DATE_TYPE** be the set of all dates.
Let **AFSC_TYPE** be the set of all possible AFSC codes.
Let **AERO_RATING_TYPE** be the set of all aero ratings.

MilitaryFaculty

$rank : RANK\_TYPE$
$branch : BRANCH\_TYPE$
$dateofrank : DATE\_TYPE$
$AFSC : AFSC\_TYPE$
$aeroratingcode : AERO\_RATING\_TYPE$
$dateofcommission : DATE\_TYPE$
$dateofseparation : DATE\_TYPE$
$manningcode :$
$DEROSdate :$

CivilianFaculty Structure Definition

Object Name: CivilianFaculty

Object Number:

Object Description: General model of a civilain faculty

Date: 08/03/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Faculty

Components: None

Context: None

Attributes:

grade	grade type	Civilian grade.
dateofgrade	date type	Date of grade.
manningcode		
DEROSdate	date type	

Constraints:

Z Static Schema:
Let **GRADE_TYPE** be the set of civilain grade types.
Let **DATE_TYPE** be the set of all dates.

CivilianFaculty

$grade : GRADE\_TYPE$
$dateofgrade : DATE\_TYPE$
$manningcode :$
$DEROSdate :$

CivilianStudent Structure Definition

Object Name: CivilianStudent

Object Number:

Object Description: General model of a civilian student.

Date: 08/03/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Student

Components: None

Context: None

Attributes:

grade	grade type	Civilian grade.
dateofgrade	date type	Date of grade.
manningcode		
DEROSdate	date type	

Constraints:

Z Static Schema:
Let **GRADE_TYPE** be the set of civilain grade types.
Let **DATE_TYPE** be the set of all dates.

CivilianStudent

$grade : GRADE\_TYPE$
$dateofgrade : DATE\_TYPE$
$manningcode :$
$DEROSdate :$

MilitaryStudent Structure Definition

Object Name: MilitaryStudent

Object Number:

Object Description: General model of a military student

Date: 07/29/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Student

Components: None

Context: None

Attributes:

rank	rank type	Military rank.
branch	branch type	Branch of service.
dateofrank	date type	Date of rank.
AFSC	AFSC type	AFSC code.
aeroratingcode	rating type	Aero rating.
dateofcommission	date type	Date of commission.
dateofseparation	date type	Date of separation.
manningcode		
DEROSdate	date type	

Constraints:

Z Static Schema:
Let **RANK_TYPE** be the set of military rank types.
Let **BRANCH_TYPE** be the set of all armed service branches types.
Let **DATE_TYPE** be the set of all dates.
Let **AFSC_TYPE** be the set of all possible AFSC codes.
Let **AERO_RATING_TYPE** be the set of all aero ratings.

MilitaryStudent

$rank : RANK\_TYPE$
$branch : BRANCH\_TYPE$
$dateofrank : DATE\_TYPE$
$AFSC : AFSC\_TYPE$
$aeroratingcode : AERO\_RATING\_TYPE$
$dateofcommission : DATE\_TYPE$
$dateofseparation : DATE\_TYPE$
$manningcode :$
$DEROSdate :$

InternationalStudent Structure Definition

Object Name: InternationalStudent

Object Number:

Object Description: General model of an international student.

Date: 08/03/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Student

Components: None

Context: None

Attributes:
WSCN WSCN type
ITO

Constraints:

Z Static Schema:
Let **WSCN_TYPE** be the set of WSCN types.
Let **DATE_TYPE** be the set of all dates.

InternationalStudent

$WSCN : WSCN\_TYPE$

$ITO :$

Course Structure Definition

Object Name: Course

Object Number:

Object Description: General model of an AFIT course class.

Date: 07/29/94
History: Thesis

Author: Capt Douglas Wu

Superclass: None

Components: None

Context: None

Attributes:

prefix	String	Course prefix.
number	integer	Course number.
section	character	Course section.
title	String	Course title.
instructor	Person pointer	Pointer to instructor.
students	Person pointer set	Set of pointers to students.
instructorcontacthr	integer	Class time.
labcontacthours	integer	Lab time.
mincredithours	integer	minimum credit hours.
maxcredithours	integer	maximum credit hours.
classstarttime	time type	Time class starts.
classendtime	time type	Time class ends.
classmeetingdays	day type set	Set of days class meets.
classroom	Room pointer	Pointer to classroom.
labstarttime	time type	Time lab starts.
labendtime	time type	Time lab ends.
labmeetingdays	day type	Day lab meets.
labroom	Room pointer	Pointer to lab.
term	term type	Term scheduled.
quarter	quarter type	Quarter.
yearprefix	Year type	Year prefix.

Constraints:

Z Static Schema:

Let **PERSON_PTR** be a pointer to a person.
Let **PERSON_PTR_SET** be the set of pointers to people.
Let **TIME** be the data type of all valid times.
Let **DAY_SET** be a set of days.
Let **ROOM_PTR** be a pointer to a room.
Let **DAY_TYPE** be a data type for days.
Let **TERM_TYPE** be a data type for academic term.
Let **QUARTER_TYPE** be a data type for academic quarter.
Let **YEAR_TYPE** be a data type for year.

Course

$prefix : String$
$number : \mathcal{N}$
$section : character$
$title : String$
$instructor : PERSON\_PTR$
$students : PERSON\_PTR\_SET$
$instructor\_contact\_hr : \mathcal{N}$
$lab\_contact\_hours : \mathcal{N}$
$min\_credit\_hours : \mathcal{N}$
$max\_credit\_hours : \mathcal{N}$
$class\_start\_time : TIME$
$class\_end\_time : TIME$
$class\_meeting\_days : DAY\_SET$
$classroom : ROOM\_PTR$
$lab\_start\_time : TIME$
$lab\_end\_time : TIME$
$lab\_meeting\_days : DAY\_TYPE$
$lab\_room : ROOM\_PTR$
$term : TERM\_TYPE$
$quarter : QUARTER\_TYPE$
$year\_prefix : YEAR_TYPE$

Book Structure Definition

Object Name: Book

Object Number:

Object Description: General model of a book class.

Date: 07/29/94
History: Thesis

Author: Capt Douglas Wu

Superclass: None

Components: None

Context: None

Attributes:

title	String	Book title.
author	String	Book author.
copyrightyear	integer	Book copyright year.
publisher	String	Book publisher.
usedbycourses	Course pointer set	Set of pointers to course.

Constraints:

Z Static Schema:
Let **COURSE_PTR_SET** be the set of pointers to courses requiring book.

Book

$title : String$
$author : String$
$copyright : \mathbb{N}$
$publisher : String$
$used\_by\_courses : COURSE\_PTR\_SET$

Building Structure Definition

Object Name: Building

Object Number:

Object Description: General model of a building class.

Date: 08/03/94
History: Thesis

Author: Capt Douglas Wu

Superclass: None

Components: None

Context: None

Attributes:

name	String	Building name.
number	integer	Building number.
rooms	Room pointer set	Pointers set to rooms in building.

Constraints:

Z Static Schema:
Let **ROOM_PTR_SET** be the set of pointers to rooms in building.

Building

$name : String$
$number : \mathcal{N}$
$rooms : ROOM\_PTR\_SET$

Room Structure Definition

Object Name: Room

Object Number:

Object Description: General model of a room class.

Date: 08/03/94
History: Thesis

Author: Capt Douglas Wu

Superclass: Building

Components: None

Context: None

Attributes:

number	integer	Room number.
size	dimension type	Room dimensions in feet.
seats	integer	Number of seats.
chalkboard	boolean	
whiteboard	boolean	
viewgraphscreen	boolean	
viewgraphprojector	boolean	
usedbycourses	Course pointer set	Set of pointers to course.

Constraints:

Z Static Schema:
Let **DIMENSION_TYPE** be a pair of integers (width and legnth) of a room.
Let **COURSE_PTR_SET** be the set of pointers to courses requiring book.

Room

$number : \mathcal{N}$
$size : DIMENSION\_TYPE$
$seats : \mathcal{N}$
$chalkboard : boolean$
$whiteboard : boolean$
$viewgraphscreen : boolean$
$viewgraphprojector : boolean$
$used\_by\_courses : COURSE\_PTR\_SET$

APPENDIX B

STARS Relational Tables provided by Katherine Hale, 16 June 1994

Primary keys are underlined and foreign keys emphasized.

Academic_Action_Valid(Academic_Action_Code, Academic_Action, Input_Date, Login_Name)
Academic_Ed_Status_Valid(Academic_Ed_Status_Code, Academic_Ed_Status, Input_Date, Login_Name)
Academic_Rank_Valid(Academic_Rank_Code, Academic_Rank, Input_Date, Login_Name)
Activity_Type_Valid(Activity_Type_Code, Activity_Type, Login_Name, Input_Date)
Address(SSAN, *Address_Type_Code*, Firm_Name_Office_Symbol, Additional_Address_Information, Street_Address, City, State_Code, Zipcode, Zipcode_Extension, *Street_Type_Code*, *Address_Room_Type_Code*, Address_Room_Type_Number, Area_Code, Phone_Number, Address_Effective_Date, DSN_Prefix, Login_Name, Revision_Name, Revision_Date, Country, Login_Date, Address_Line_1, Address_Line_2, Address_Line_3, *Country_Code*)
Address_Room_Type_Code_Valid(Address_Room_Type_Code, Address_Room_Type_Code_Des, Input_Date, Input_Name, Revision_Date, Revision_Name)
Address_Type_Valid(Address_Type_Code, Address_Type, Input_Date, Login_Name)
Admission_Type_Valid(Admission_Type_Code, Admission_Type, LoginName, InputDate)
AFIT_Degree_Valid(AFIT_Degree_Code, AFIT_Degree, Input_Date, Login_Name, *Type_Degree_Code*, ABET_Accredited_Indicator)
AFSC_Valid(AFSC_Code, AFSC, Enlisted_Indicator, Input_Date, Login_Name)
Award_Valid(Award_Code, Award, Input_Date, Login_Name)
Box_Number_Valid(Box_Number, Login_Name, Input_Date)
Branch_Service_Valid(Branch_Service_Code, Branch_Service, Input_Date, Login_Name, MPC_Branch_Service_Abbrev)
Building_Valid(Building_Code, Building, Input_Date, Login_Name)
Career_Pointer_Valid(Career_PointerCode, Career_Pointer, Input_Date, Login_Name)
Countries_Attending_AFIT (AFSAT_Country_Code, Country, Country_Count)
Country_Valid(Country_Code, Country, Login_Name, Input_Date)
Course_Books(Course_Prefix_Code, Course_Number, Course_Section, Book)
Course_Books_Unofficial(Course_Prefix_Code, Course_Number, Course_Section, Book)
Course_Coreqs_Unofficial(Course_Number, Course_Prefix_Code, Course_Section, Coreq_Effective_Date, Coreq_Text, Coreq_Text_Line_Number)
Course_Corequisites(Course_Number, Course_Prefix_Code, Course_Section, Coreq_Effective_Date, Coreq_Text, Coreq_Text_Line_Number)
Course_Notes_Unofficial(Course_Number, Course_Prefix_Code, Course_Section, Note_Code)
Course_Offerings(Course_Prefix_Code, Course_Number, Course_Section, Course_Title, Projected_Sections, *Schedule00Term_Code*, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*, Exam_Indicator, Instructor_Contact_Hours, Lab_Contact_Hours, Min_Credit_Hours, Max_Credit_Hours, *Course_Offering_Status_Code*, *Course00Department_Code*, Remarks, Input_Date, Login_Name)
Course_Offerings_Remarks(*Schedule00Term_Code*, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*, *Course00Department_Code*, Remark, Remark_Sequence_Number)
Course_Offering_Instructor(Course_Prefix_Code, Course_Number, Course_Section, Faculty_SSAN, *Course_Taught00Term_Code*, Instructor_Load, *Course_Taught00Quarter_Code*, *Course_Taught_Year_Prefix*, Enrollment_Limit)
Course_Offer_Status_Valid(Course_Offering_Status_Code, Course_Offering_Status, Input_Date, Login_Name)

Course_Prerequisites(Course_Number, Course_Prefix_Code, Course_Section, Prereq_Effective_Date, Prereq_Text, Prereq_Text_Line_Number)

Course_Prereq_Sub_Course(Course_Prefix_Code, Course_Number, *Prereq00Course_Number*, *Prereq00Course_Prefix_Code*, *Prereq_Sub00Course_Prefix_Code*, *Prereq_Sub00Course_Number*, Prereq_Sub_Effective_Date)

Course_Room_Requirements(Course_Number, Course_Prefix_Code, Course_Section, *Room_Requirement_Code*)

Course_Unofficial(*Activity_Type_Code*, *AFIT_School_Code*, *Course00Department_Code*, Course_Effective_Date, *Course_Level_Code*, Course_Number, Course_Prefix_Code, Course_Section, Course_Title, *Credit_Check_Code*, Exam_Indicator, Individual_Contact_Hours, Input_Date, Instructor_Contact_Hours, Login_Name, Max_Credit_Hours, Max_Enrollment_Limit, Min_Credit_Hours, Min_Enrollment_Limit, Releasibility_Indicator, *Special_Grading_Code*, *Wait_List_Code*, *Grading00Department_Code*, Lab_Contact_Hours, Full_Course_Title)

Degree_Candidates(SSAN, Degree_Candidate_Date, *AFIT_Degree_Code*)

Department_Valid(Department_Code, Department, *Department_Head_SSAN*, *Directorate00Department_Code*, Input_Date, Login_Name, *RCCC_Code*, DSN_Phone)

Dependent_Children(SSAN, Dependent_Child_Birth_Date, Child_Last_Name, Child_First_Name, *Dependent_At_AFIT_Indicator*, *Child00Sex_Code*)

Dependent_Information(SSAN, Number_Children, Sngle_Dep_Chldrn_Indicator, Dependent_at_AFIT_Indicator)

Distribution_Valid(Distribution_Code, Distribution, Input_Date, Login_Name)

Duty_History(SSAN, Duty_Effective_Date, *CBPO_Code*, *MAJCOM_Code*, *Org_Loc_Code*, Duty_Title, *Base_Code*, *Duty00AFSC_Code*, Duty_Phone, DSN_Phone, DSN_Prefix, Input_Date, Login_Name)

Edplan_Desc(SSAN, *Career_Pointer_Code*, Description, Description_Line_Number)

Edplan_Errors(Course_Prefix_Code, Course_Number, Course_Section, *Schedule00Term_Code*, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*)

Education_History(SSAN, Quality_Points, Total_Credit_Hours, *MPC_School_Code*, *ASC_Code*, *Method_Of_Obtainment_Code*, *Academic_Ed_Status_Code*, Input_Date, Operators_Initials, Login_Name, Last_Year_Attended, *Ed_level_Code*, ABET_Accredited_Indicator, Ed_History_Remarks, *Work_ID_Processed_Code*, *Trnscrpt00Career_Pointer_Code*, *Type_Degree_Code*, *Duty_Location_Code*, Degree_CUM_GPA, Degree_Title)

Ed_Level_Valid(Ed_Level_Code, Ed_Level, Input_Date, Login_Name)

Ethnic_Group_Valid(Ethnic_Group_Code, Ethnic_Group, Login_Name, Input_Date)

Evaluation_By_School(SSAN, Admitted_Indicator, Evaluation_Result_Remark, Evaluation_Forwarded_Date, *Forwarded_To00Department_Code*, Evaluation_Returned_Date)

Evaluation_Status_Valid(Evaluation_Status_Code, Evaluation_Status, Login_Name, Input_Date)

Exam_Schedule(Course_Number, Course_Prefix_Code, Course_Section, *Exam00Term_Code*, Exam_Date, Exam_Time, *Exam00Room_Code*, *Exam00Building_Code*, *Exam00Quarter_Code*, *Exam_Year_Prefix*)

Exception(SSAN, *Address_Type_Code*, Address_Line1, Address_Line2, Address_Line3)

Faculty_Civilian(SSAN, *Occupation_Series_Code*)

Faculty_History(SSAN, Academic_Rank_Date, *Academic_Rank_Code*, Academic_Step)

Faculty_Type_Valid(Faculty_Type_Code, Faculty_Type, Input_Date, Login_Name)

Fitness_Category_Valid(Fitness_Category_Code, Fitness_Category, Input_Date, Login_Name, Measurement_Unit)

Fitness_Performance(SSAN, *Fitness_Category_Code*, Elapsed_Time, Trial_Date, Input_Date, Login_Name, Distance)

Fitness_Standards_Valid(Low_Age_Range, High_Age_Range, Fitness_Category_Code, *Sex_Code*, Elapsed_Time, Input_Date, Login_Name, Distance)

Funding(SSAN, Disbursement_Date, *Funding_Code*, Funding_Amount)

Funding_Valid(Funding_Code, Funding, Input_Date, Login_Name)

Grade_Change_History(SSAN, Course_Prefix_Code, Course_Number, Course_Section, *Prior00Grade_Code*, *Schedule00Term_Code*, Grade_Effective_Date, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*)

Grade_History(SSAN, Approval_Code, Approval_Date, Career_Pointer_Code, Credit_Hours, Earned_Hours_Indicator, GPA_Indicator, *Course_Prefix_Code, Course_Number,* Input_Date, *Course_Section, Schedule00Term_Code, Grade_Code, Grade_Type_Code*, Login_Name, *Prior00Grade_Code, Grade_Effective_Date, Schedule00Quarter_Code, Schedule_Year_Prefix*)

Grade_Type_To_Grade_Valid(Grade_Type_Code, Grade_Code, Login_Name, Input_Date)

Grade_Type_Valid(Grade_Type_Code, Grade_Code, Login_Name, Input_Date, GPA_Indicator, Earned_Hours_Indicator)

Grade_Valid(Grade_Code, Grade, Input_Date, Login_Name, Grade_Points, GPA_Indicator, Earned_Hours_Indicator)

Graduation_Attendees(SSAN, *Graduation00Term_Code, Graduation00Quarter_Code, Graduation_Year_Prefix*)

Graduation_Date(Graduation_Date, Graduation_Place, *Graduation00Term_Code*, Login_Name, Input_Date, *Graduation00Quarter_Code*, *Graduation_Year_Prefix*)

Grad_Audit(Grad_Audit_Code, Grade_Audit, Input_Date, Login_Name)

Grad_Status_Valid(Grade_Status_Code, Grade_Status, Login_Name, Input_Date)

ID(ID)

Intl_Sponsor(Intl_Sponsor_SSAN, Intl_Sponsor_Occupation, *ACComp_Preference_Code*, Avail_Start_Date, Avail_End_Date)

Intl_Student(SSAN, WSCN, ITO, Case_Number, DLI_Req_Indicator, DLI_Indicator, Evaluation_Request_Date, *Requested00Program_Code*, Eval_Forward_Date, *ForwardTo00Department_Code*, Eval_Returned_Date, *Admission_Status_Code*, Eval_Remarks, Country_Notified_Date, AFSAT_Notified_Date, AFSAT_Quota_Indicator, *First_Sponsor_SSAN*, *Second_Sponsor_SSAN*, *Source_Of_Funds_Code*, *AFSAT_Country_Code*)

Invention(Invention, Inventor, Year, Nation_Code)

IP_Activity_Objectives(IP_Activity_Code, IP_Objective_Code)

IP_Activity_Valid(IP_Activity_Code, IP_Activity, Login_Name, Input_Date)

IP_Objective_Valid(IP_Objective_Code, IP_Objective, Input_Date, Login_Name)

IP_Resource(IP_Activity_Code, Activity_Address_1, Activity_Address_2, Activity_Address_3, Activity_City, Activity_State, Activity_Zipcode, Activity_POC, Activity_POC_Phone, Availability_Comment, Activity_Evaluation, POC_Area_Code, Firm_Name_Office_Symbol, Additional_Address_Information, Street_Address, *Street_Type_Code*, *Address_Room_Type_Code*, *Address_Room_Type_Number*, Activity_Zipcode_Extension, Login_Name, Login_Date, Revision_Name, Revision_Date)

Job_Title_Valid(Job_Title_Code, Job_Title, Login_Name, Input_Date)

Language_Valid(Language_Code, Language, Input_Date, Login_Name)

Leader_Valid(Leader_Code, Leader, Login_Name, Input_Date)

Letter_Status(Letter_Status_Code, Letter_Status, Login_Name, Input_Date)

Locker_Valid(Locker_Number, Combination, Locker_Size, Login_Name, Input_Date)

LS_Part_Time(SSAN, *Pt_LS_Student00Program_Code*)

Mail_Building(AFIT_School_Code, Building_Code, Login_Name, Input_Date)

MAJCOM_Valid(MAJCOM_Code, MAJCOM_Abbrev, MAJCOM, Input_Date, Login_Name)

Majors(SSAN, Career_Pointer_Code, Major, Login_Name, Input_Date)

Majors_Valid(Major, Login_Name, Input_Date)

Manning_Code(Manning_Code, Manning, Login_Name, Input_Date, *Branch_Service_Code*)

Marital_Status_Valid(Marital_Status_Code, Marital_Status, Input_Date, Login_Name)

Method_Of_Obtainment(Method_Of_Obtainment_Code, Method_Of_Obtainment, Input_Date, Login_Name)

Name_History(SSAN, Name_Change_Date, First_Name, Last_Name, Middle_Initial, Name_Suffix, Name_Prefix, Login_Name, *Marital_Status_Code*)

Newsletter(Addressee, Address_Line_1, Address_Line_2, Address_Line_3, City, State_Code, Zipcode, Zipcode_Extension, Country_Code)
New_Table(SSAN, AFSC)
Notes_Code(Note_Code, Note, Login_Name, Input_Date)
Occupation_Series_Valid(Occupation_Series_Code, Occupation_Series, Login_Name, Input_Date)
Options(SSAN, Career_Pointer_Code, Options, Login_Name, Input_Date)
Options_Valid(Options, Login_Name, Input_Date)
Person(SSAN, Grade_Rank_Abbrev, Name_Prefix, Name_Suffix, First_Name, Last_Name, Middle_Initial, Birth_Date, *Sex_Code*, *Race_Code*, *Marital_Status_Code*, *Religion_Code*, Blue_Chip_Indicator, Aka_FName, Aka_LName, Prior_AFIT_Months, TAFMS_Date, *Ethnic_Group_Code*, *Aero_Rating_Code*, *Manning_Code*, DEROS_Date, Separation_Date, *Commission_Code*, Grade_Rank_Date, *Citizenship00Country_Code*, *Department_Code*, Duty_Title, Duty_Phone, Duty_Area_Code, Badge_Number, *Branch_Service_Code*, Login_Name, Input_Date, Duty_Phone_Ext)
Person_Job_Title(SSAN, Job_Title_Code)
Planes_Flown(SSAN, Plane_Name)
POC_Address(*Civilian_Institution_Code*, *School_In_Civ_Ins_Code*, *POC_Type_Code*, POC_Type_Seq, Department_Name, Hospital_Firm_Name, Street_Address, *Street_Type_Code*, *Address_Room_Type_Code*, *Room_Type_Number*, City, *State_Code*, Zip, Zip_Extension, *Country_Code*, Department_Entering_POC, Dept_Entering_Address, Dept_Entering_POC, Login_Name, Input_Date)
Prof_Military_Ed Valid(PME_Level_Code, PME_Description, Login_Name, Input_Date)
Program_Advisor(Program_Code, *Program_Graduation00Term_Code*, Class_Code, Program_Year_Prefix, Program_Advisor_SSAN)
Program_Grad_Adit(Audit00Program_Code, Audit_Effective_Date, *Course_Number*, *Course_Prefix_Code*, *Grad_Audit_Code*)
Program_Option(Program_Code, *Option00Program_Code*, Program_Option, Input_Date, Login_Name)
Program_Plan(Program_Code, *Program_Graduation00Term_Code*, *Class_Code*, *Program_Year_Prefix*, *Course_Section*, *Course_Number*, *Course_Prefix_Code*, Credit_Hours, *EdPlan00Term_Code*, *EdPlan00Quarter_Code*, *EdPlan_Year_Prefix*, Input_Date, Login_Name, *Course_Type_Code*)
Program_Sections(*Section_Nu*mber, *Program_Code*, *Program_Graduation00Term_Code*, *Class_Code*, *Program_Year_Prefix*)
Program_Sequences(Program_Code, Class_Code, Year_Prefix, *Program_Sequence_Code*, *Advisor00SSAN*, Input_Date, Login_Name)
Program_Seq_Courses(Program_Sequence_Code, *Prog_Seq_Courses00Term_Code*, *Prog_Seq_Courses00Quarter_Code*, *Prog_Seq_Courses00Year_Prefix*, *Course_Prefix_Code*, *Course_Number*, *Course_Section*, *Grade_Type_Code*, Credit_Hours)
Program_Std_Sections(SSAN, Section_Number)
Race_Valid(Race_Code, Race, Login_Name, Input_Date)
Rank_History(SSAN, Grade_Rank_Date, Grade_Rank_Abbrev, Login_Name, Input_Date, Manning_Code, *Branch_Service_Code*)
Rank_Valid(Pay_Type_Code, Pay_Level_Code, NCO_Indicator, *Rank_Code*, Rank, Branch_Service_Code, Login_Name, Input_Date, *MPC_Code*, Printed_Rank)
Recall_Roster(SSAN, Home_Phone_Number, *Next_In_Chain_SSAN*)
Registration_Verification (SSAN, Term_Code, Quarter_Code, Year_Prefix, Registration_Notice)
Religion(Religion_Code, Religion, Input_Date, Login_Name)
Reselection(Reselection_Code, Reselection, Input_Date, Login_Name)
Resident_Student(SSAN, *Academic_Action_Code*, Overdue_Indicator, *Classification_Code*, Part_Record_Indicator, Admin_Hold_Indicator, *Major00ASC_Code*, *Program_Graduation00Term_Code*, *Class_Code*, *Program_Year_Prefix*, *Selected_Type_Code*, *AFIT_Degree_Code*, *Graduation00Term_Code*, *Graduation00Quarter_Code*, *Graduation_Year_Prefix*, *Grad_Status_Code*, Departure_Date, Box_Number, Card_Number, Encoded_Card_Number, Library_Number, Locker_Number, Admit_Date,

Student_Sponsor_SSAN, Entry_Year_Prefix, *Admission_Type_Code*, Admission_Action_Code, *Career_Pointer_Code*, *Gaining00AFSC_Code*, *Faculty_Advisor_SSAN*, *Registration00Department_Code*, *Program_Effective00Term_Code*, *Effective00Quarter_Code*, *Effective_Year_Prefix*, *Leader_Code*, Program_Section_Number, Gain00MAJCOM_Abbrev, Gain00Duty_Station, Losing00MAJCOM_Abbrev, PSE_Code)

Resident_Student_Default(Ed_Level_Code, Classification_Code, Grad_Status_Code, Admission_Type_Code, Admin_Hold_Indicator, Part_Record_Indicator, Academic_Action_Code, Admission_Action_Code, Overdue_Indicator, Career_Pointer_Code)

Room(*Building_Code*, Room_Code, Room_Size, Room_Remark, Input_Date, Login_Name)

Room_Requirement(Room_Code, *Building_Code*, Room_Requirement_Code)

Room_Requirements(Room_Requirement_Code, Room_Requirement, Login_Name, Input_Date)

Room_Schedule(Building_Code, Room_Code, Room_Start_Time, Room_End_Time, Room_Schedule_Date, Room_Schedule_Remark, Room_Schedule_Contact)

Schedule(*Activity_Type_Code*, *AFIT_School_Code*, Course_End_Date, Course_End_Time, *Course_Level_Code*, Course_Number, Course_Prefix_Code, Course_Section, Course_Start_Date, Course_Start_Time, Course_Title, *Course00Department_Code*, *Cedit_Check_Code*, *Day_Code*, Exam_Indicator, Individual_Contact_Hours, Instructor_Contact_Hours, Max_Credit_Hours, Max_Enrollment_Limit, Min_Credit_Hours, Min_Enrollment_Limit, Releasibility_Indicator, *Schedule00Building_Code*, *Schedule00Room_Code*, *Schedule00Term_Code*, *Special_Grading_Code*, *Wait_List_Code*, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*, *Grading00Department_Code*)

Schedule_Mandatory(Mandatory_Code, Mandatory, Login_Name, Input_Date)

Schedule_Notes(Course_Number, Course_Prefix_Code, Course_Section, Note_Code, *Schedule00Term_Code*, *Schedule00Quarter_Code*, *Schedule_Year_Prefix*)

Schedule_Room_Requirements(Course_Number, Course_Prefix_Code, Course_Section, Room_Requirement_Code, Schedule00Term_Code, Schedule00Quarter_Code, Schedule_Year_Prefix)

Schools(MPC_School_Code, School_Abbrev, School, State_Code, City, Input_Date, Login_Name, *Country_Code*)

School_In_Civ_Ins(Civilian_Institution_Code, School_In_Civ_Ins_Code, School_Name, Login_Name, Input_Date)

Section_Leaders(SSAN, Leader_Code, *Program_Code*, *Class_Code*, *Program_Section_Number*)

Security_Class_Valid(Security_Class_Code, Security_Class,Login_Name, Input_Date)

Security_Clearance_Valid (Security_Clearance_Code, Security_Description, Input_Date, Login_Name)

Selected_Comments(SSAN, Selected_Comment)

Selected_Projection(SSAN, *Gain00MAJCOM_Code*, Gain00MAJCOM_Abbrev, Gain_MAJCOM_Supervisor, Gain_MAJCOM_Supervisor_Phone, Gain_MAJCOM_DSN_Prefix, Gain_MAJCOM00Department_Code, Position_Number_Projected)

Selected_Type(Selected_Type_Code, Selected_Type, Input_Date, Login_Name)

Sex(Sex_Code, Sex, Input_Date, Login_Name)

Source_Of_Funds_Valid(Source_Of_Funds_Code, Source_Of_Funds, Login_Name, Input_Date)

Special_Grading(Special_Grading_Code, Special_Grading, Input_Date, Login_Name)

Special_Sched_Request(Approval_Code, Approval_Date, Course_Number, Course_Prefix_Code, Course_Section, Day_Code, *Faculty_SSAN*, Mandatory_Code, Priority, Request_Alternative_Number, Request_End_Time, Request_Justification, Request_Start_Time, Request00Activity_Type_Code, Request00Building_Code, Request00Room_Code, Request00Term_Code, Request00Quarter_Code, Request_Year_Prefix)

Sponsors_Country_Preference(SSAN, *Preferred00Country_Code*)

Status_Change(Status_Change_Code, Status_Change, Input_Date, Login_Name)

Street_Type_Code_Valid(Street_Type_Code, Street_Type_Code_Description, Input_Date, Input_Name, Revision_Date, Revision_Name)

Student_Duty_History(SSAN, Duty_Title, *Duty00AFSC_Code*, Duty_Organization, Duty_Station, Duty_Assigned_Date, Login_Name, Duty_Sequence_Number)

Student_Sequences(SSAN, *Program_Sequence_Code*)
Suspense_Calender(Suspense_Code, From00Department_Code, To00Department_Code, Suspense_Date, Term_Code, Quarter_Code, Year_Prefix)
Suspense_User_Class(Suspense_Code, Security_Class_Code, Username)
Suspense_Valid(Suspense_Code, Suspense, Login_Name, Input_Date)
Swim_Times(SSAN, Effective_Date, Swim_Time)
TDY_Attendees(SSAN, Left_For_TDY_Date, Returned_From_TDY_Date, *TDY_Destination_Code*, TDY_Purpose)
TDY_Destination_Valid(TDY_Destination_Code, TDY_Destination, Login_Name, Input_Date)
Terms(Term_Code, Term, Login_Name, Input_Date, Quarter_Code, Class_Code, Year_Prefix)
Term_Date(Term_Code, Term_Start_Date, Term_End_Date, Input_Date, Login_Name, Regs_Cutoff_Date, Quarter_Code, Year_Prefix, Offering_Cutoff_Date, Grad_Stud_Grades_Cutoff_Date, Nongrad_Grades_Cutoff_Date, Thesis_Cutoff_Date, School_Schedule_Start_Date, School_Schedule_End_Date)
Term_Entry(SSAN, *Entry00Term_Code*, *Entry00Quarter_Code*, *Entry_Year_Prefix*, *Admission_Type_Code*, *Admission_Action_Code*)
Term_Entry_History(SSAN, *Entry00Term_Code*, *Entry00Quarter_Code*, *Entry_Year_Prefix*, *Admission_Type_Code*, *Admission_Action_Code*)
Test_Type(Test_Type_Code, Test_Type, Login_Name, Input_Date, Maximum_Score)
Thesis_Sponsor(Thesis_Sponsor_Code, Thesis_Sponsor, Sponsor_POC, DSN_Prefix, Thesis_Sponsor_Phone)
Thesis_Title(Thesis_Diss_Key, Thesis_Title, Thesis_Sequence_Number)
Wait_List(SSAN, *Course_Prefix_Code*, *Course_Number*, *Course_Section*)
Wait_List_Types(Wait_List_Code, Wait_List, Login_Name, Input_Date)
Waived_Course(SSAN, *Waived00Course_Prefix_Code*, *Waived00Course_Number*, *Waived00Grade_Code*, Waived_Date)

APPENDIX C

LIST OF ABBREVIATIONS

AFIT - Air Force Institute of Technology

AFITSIS - Air Force Institute of Technology Student Information System

BNF - Backus Naur Form

CAD - Computer-Aided Design

CASE - Computer-Aided Software Engineering

CCQ - AFIT orderly room

DAGSI - Dayton Area Graduate Studies Institute

DBMS - Database Management System

DFD - Data Flow Diagram

ISA - International Student Affairs

LOC - lines of code

ODMG-93 - Object Database Management Group

ODBMS - Object Database Management System

ODL - Object Data Language

OID - Object Identifier

OMT - Object Modeling Technique

OODBMS - Object-Oriented Database Management System

OQL - Object Query Language

QUEST - Quota Education and Selection Transactions

RDBMS - Relational Database Management System

SQL - Structured Query Language

STARS - Student Tracking and Registration System

Bibliography

1. Ahmed, Shamin, Albert Wong, Duvvuru Sriram, and Robert Logcher. "Object-Oriented Database Management Systems for Engineering: A Comparison," Journal of Object-Oriented Programming, June 1992. pp. 27-44.

2. Atwood, Thomas. "The Case for Object-Oriented Databases," IEEE Spectrum, February 1991, pp. 44-47.

3. Atwood, Thomas. "ODMG-93: The Object DBMS Standard, part 2," Object Magazine, January 1994, pp. 32-37.

4. Bertino, Elisa and Lorenzo Martino. "Object-Oriented Database Management Systems: Concepts and Issues," IEEE Computer, April 1991. pp. 33-47.

5. Cattell, R.G.G., Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis, Drew Wade. The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, Inc, 1994.

6. Cerney, Caroline. AFIT/SCQ Personal Interview, April 1994.

7. Coad, Peter and Edward Yourdon. Object-Oriented Design. Yourdon Press, 1991.

8. Cohen, Norman. Ada as a Second Language. McGraw-Hill Book Company, 1986.

9. Davis, Richard. Thesis Projects in Science and Engineering. St. Martin's Press, 1980.

10. Feldman, Michael, and Elliot Koffman. Ada: Problem Solving and Program Design. Addison-Wesley Publishing Company, 1993. pp. 1-795.

11. Hale, Katherine. AFIT/SCQ Personal Interview, April 1994.

12. Hartrum, Thomas and Paul Bailor. A Formal Extension to OOA (Unpublished). Air Force Institute of Technology, April 1993. pp. 1-70.

13. Joseph, John, Satish Thatte, Craig Thompson, and David Wells. "Object-Oriented Databases: Design and Implementation". Proceedings of the IEEE, Vol. 79, No. 1, January 1991. pp. 42-64.

14. Kim, Won. "A New Database For New Times". Datamation: 35-42. 15 January 1990.

15. Kim, Won. "Defining Object Databases Anew". Datamation: 33-36. 1 February 1990.

16. Korth, Henry, and Abraham Silberschatz. Database System Concepts, Second Edition. McGraw-Hill, Inc., 1991.

17. Nyberg, Karl. The Annotated Ada Reference Manual, 2nd Edition. Grebyn Corporation, 1991. pp. 1-500.

18. Rasmus, Daniel. "Relating to Objects". Byte: 161-165. December 1992.

19. Rumbaugh, James and others. Object-Oriented Modeling and Design. Prentice-Hall, 1991.

20. Itasca LISP API User Manual for Release 2.2. Itasca Systems, Inc., 1993.

21. STARS User's Manual (AFIT Database). Systems Research Laboratories, 1987.

Vita

Captain Douglas James Wu was born in Springfield, Vermont on 22 July 1967. He graduated from Black River High School in Ludlow, Vermont in 1985. He attended the Military School of Vermont, Norwich University, where he was awarded the degree of Bachelor of Science in Electrical Engineering in May of 1989. After commissioning through Reserve Officers Training Corps, he was assigned to the Ballistic Missile Organization, Norton AFB, San Bernadino, CA where he was the Intercontinental Ballistic Missile Trainer Software Project Manager for the Rapid Execution and Combat Targeting modification program. In May 1993 Captain Wu entered the Air Force Institute of Technology as a Masters candidate in computer systems.

Permanent address: 48 Andover Street
Ludlow, Vermont 05149

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
The Re-Engineering of the Air Force Institute of Technology Student Information System

5. FUNDING NUMBERS

6. AUTHOR(S)
Douglas J. Wu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-27

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research describes the design and implemetation issues associated with re-engineering the Air Force Institute of Technology Student Information System (AFITSIS). Currently, AFITSIS executes on aging relational database technology and has unfriendly user interface mechanisms. The two research objectives met were to research current AFITSIS requirements, design, and implementation, and use object-oriented methods to design an alternative implementation based on proposed object database management system standards. This research explores how AFITSIS performance and capabilities might be enhanced by taking advantage of new object-oriented software engineering techniques. One of the primary benefits of this research is a detailed object modeling technique analysis and design that may be used as a foundation for upgrading the current AFITSIS.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AFITSIS, AFIT Student Information System, information system design, OODBMS Design, ODBMS Design, Database OMT	142
	16. PRICE CODE

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