USING FORMAL METHODS TO DESIGN AND IMPLEMENT AN OBJECT-ORIENTED UNIVERSITY SPORTS CENTER'S INFORMATION MANAGEMENT SYSTEM

By

YI XIE

Bachelor of Engineering

Computer Institute, Beijing Polytechnic University

Beijing, P. R. China

1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 1997

USING FORMAL METHODS TO DESIGN AND IMPLEMENT AN OBJECT-ORIENTED UNIVERSITY SPORTS CENTER'S INFORMATION MANAGEMENT SYSTEM

Thesis Approved:

H. Lu
Thesis Advisor

J Chandler

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEGMENTS

I sincerely thank my graduate advisor Dr. Huizhu Lu for the guidance, help, encouragement and time she has given me toward the completion of my thesis work. I would like to express my sincere thanks to Dr. J. P. Chandler and Dr. K. M. George for serving on my committee. Their guidance, encouragement and friendships are very helpful to improve the quality of this thesis.

My respectful thanks go to my parents Mr. Jinlai Xie and Mrs. Zhaohua Xie, and my husband, Sheng Xu, for the love, encouragement and confidence they have given me.

I would also like to express my appreciation to all those people who have contributed by giving many valuable suggestions.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Formal Methods

The goal of the software engineering process is to create high quality software. High quality software should be maintainable, reliable, and efficient, and should offer an appropriate user interface [Bruno, 1995]. In recent years, much research in software engineering has been focused on how to produce high quality software quickly and economically.

Two crucial stages in the software engineering process are software specification and verification. The software specification describes the problem which is to be solved and makes statements about what the solution of the problem should be like. The features that a good software specification should display are clear and unambiguous, accurate and complete. The software verification is actually the answer to the question "are we making the product right ?". The software specification and verification could be based on natural languages such as English, or on computer languages such as ADA or Pascal. The specifications which are based on these languages are informal. The word "informal" means that the ways in which things are described depend to at least some extent on a common understanding of what is written among those people reading the statement. If that assumed common understanding is not present, then the statements will become

either meaningless, ambiguous, or incorrect. Formal methods can be very helpful in solving this problem [Scharbach, 1988].

Formal methods are the mathematical foundation for the specification, implementation, and verification of computer systems. The mathematical notation in formal methods can standardize the software specification and design. Formal methods consist of two parts: formal specification and verified design. Formal specification uses the notation which is derived from formal logic to describe assumptions about the world in which a system will operate, requirements that the system is to achieve and a design to meet those requirements. Specifications in formal methods can provide a direct simulation of the system behavior and this simulation can be passed to the clients so that developers can get some early feedback which can be compared with user requirements for verification. In this way, a formal specification can make the software easy to maintain and reduce the risk of a piece of software not meeting its requirements. Formal methods are distinguished by their specification languages. These specification languages exploit representations with formally defined semantics and they can describe abstractly and independently the details of implementation of the desired functional behavior of a system to be developed.

Although formal methods have been studied in academia for a long time and some examples of industrial applications have been reported in the literature, a recent survey of twelve applications of formal methods showed that people are still at the early stages of using formal methods in industry [Berg, 1982]. The principal arguments against the use of formal methods are those based on the cost of the projects, the time taken to develop software and the availability of qualified and experienced staff [Ford, 1993]. The factors

which need to be balanced when deciding whether or not to use formal methods are shown in Figure 1.

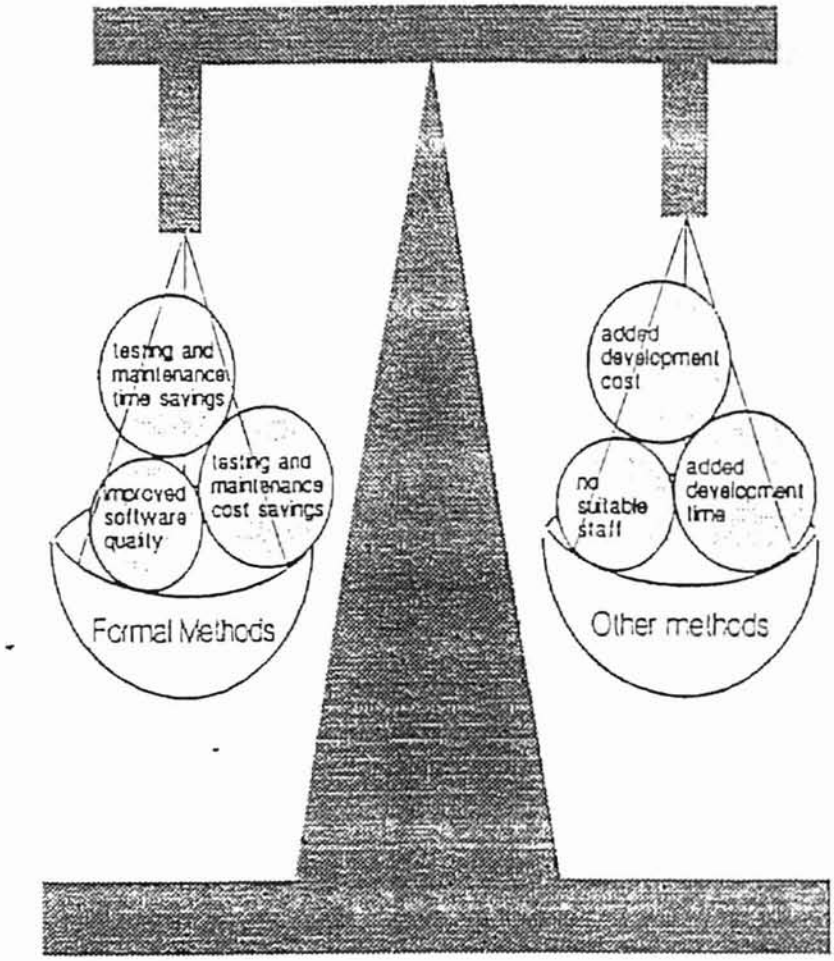

Figure 1. Factors to be balanced when deciding whether or not to use formal methods [Ford, 1993].

Z and Object-Z

As mentioned before, formal methods are distinguished by their specification languages. The Z specification language is one of the most well developed specification languages. It was developed in the late 70's and early 80's by Jean-Raymond Abrial et al. of Programming Research Group in Oxford University [Spivey, 1988].

The Z specification language is based on the mathematical disciplines of first-order logic and set theory. One of its key features is the notion of a "schema". A schema consists of a collection of named objects with a relationship specified by several axioms. Z provides notations for defining schemas and later combining them by using schema calculus, so that the specifications of sizable software systems can be built up in stages. We will briefly introduce the basic concepts of Z and the notations in Z later. Schemas can have generic parameters, and there are operations in Z for creating instances of generic schemas. Schemas are used to describe all aspects of the system under development, such as the states it can occupy, the transitions it can make from one state to another, and even the relationship between one view of the state and another as we transform the specification into a design for implementation.

The formal semantics of the Z notation is favored for several reasons. The first is that the compactness and regularity of the formulae make them easy to manipulate algebraically in a way that English text is not. The second is its consequences for the practice of specification. Formal semantics provide a foundation for a logical calculus for reasoning about specifications and deriving consequences from them. Deriving consequences from a specification is an important aid in checking that a specification

captures a customer's requirements correctly, and helps in validating proposed implementations. Finally, formal semantics also provides a view of specification which abstracts from inessential details of syntax and presentations. Z is a necessary and relatively successful attempt to devise a notation for building models of software systems and for proving that programs meet their specifications. Let us look for an example at the comparison between software engineering and civil engineering. You can not imagine that civil engineers would build a bridge by specifying the problem in natural language instead of using sophisticated mathematical techniques. While civil engineers use structural mechanics and dynamics in building a bridge, software engineer use Z which includes the knowledge of discrete mathematics, set theory and logic in specifying computer systems.

Since a number of different styles of mathematical specification are gaining popularity, it is worthwhile to compare Z with some of these [Berg, 1982]. These different styles are divided into model-oriented methods, where the aim of a specification is to construct an abstract model of the information system being specified, and property-oriented or algebraic methods, where the aim is to describe a system in terms of its desired properties, without constructing an explicit model. Among the model-oriented methods are Z and VDM (Vienna Development Method). The prominent property-oriented methods are Clear, OBJ and ACT ONE. Actually, the distinction between model-oriented and property-oriented methods is not so clear-cut. In practice, Z specifications often describe certain aspects of systems by giving axioms which must be satisfied by the system, and this amounts to property-oriented specifications. Property-oriented

specifications often describe a collection of basic data-types in a property-oriented way, then use these to build a model of the system being specified.

Object orientation is a way of structuring software and promises many benefits [Booch, 1991]. An object can be regarded as something that encapsulates a piece of a state together with some behavior such as the operations that access and modify that state. Objects provide a way to structure a system specification by partitioning an otherwise global state space into meaningful chunks. In addition objects are instances of classes, which are arranged in inheritance hierarchies. Classes can inherit properties from their parents, as well as define new states and behaviors of their own. Inheritance is an abstraction mechanism that can be used to structure a specification. Properties of similar classes are described once, in the specification of a common superclass, and hence complex classes can be specified incrementally. These object oriented concepts can be used to provide a structure for Z specifications, and this results in Object-Z. Object-Z is an extension of Z in which the existing syntax and semantics of Z are retained but new constructs are introduced to facilitate specification in an object-oriented style. This enhanced structuring improves the readability of large specifications. It also enables the possibility of modular verification and refinement. Object-Z has features such as inheritance, composing classes, adding a constant or a state invariant, modifying and redefining an operation, etc.

Microsoft Visual Basic 4.0, GUI and OOP

Visual Basic was first released by Microsoft in 1991. At that time, it was the most exciting computer language product to hit the market in a while. It is an easy-to-use, yet extraordinarily powerful tool for developing Windows applications. The latest version of Visual Basic, Microsoft Visual Basic 4.0, adds many new features, such as the ability to build 32-bit executables for both Windows 95 and Windows NT.

Graphical User Interfaces, or GUIs, have revolutionized the microcomputer industry. Instead of the cryptic C:> prompt that DOS users have long seen, users are presented with a desktop filled with icons and with programs that use mice and menus. Users can spend more time mastering the application and less time worrying about which keystrokes do what within menus and dialog boxes. Beginning users seem to like GUIs very much. Windows programs are expected to be based on the GUI model to have the right look and feel. You will want a tool to develop GUI-based applications efficiently when you need to develop programs for any version of Windows.

There were no such tools for a long time before Visual Basic was introduced in 1991. Since then, Visual Basic has made the programming for Windows not only more efficient but also more fun. The latest Microsoft Visual Basic 4.0 has many advantages over the first three versions of Visual Basic:

- We have the ability to generate 32-bit applications for both Windows 95 and Windows NT.
- We can take advantage of Microsoft's OLE (Object Linking and Embedding) technology.

- We can build programs using some of the techniques of object-oriented programming.
- We have the ability to extend the Visual Basic programming environment.
- We have conditional compilation which allows you to do multiplatform development more easily.

The general steps to design a Visual Basic application are the following. First of all, we customize the windows that the user sees. Secondly, we decide what events the controls on the window should recognize. Finally, we write the event procedures for those events and the subsidiary procedures that make those event procedures work.

Visual Basic 4.0 is the first version of Visual Basic that gives people access to some of the power and advantages of object-oriented programming (OOP), although it is not a fully OOP language like C++ or Delphi. OOP seems to be the dominant programming paradigm these days, having replaced the structured programming techniques that were developed in the early 70's.

Several important concepts in OOP are classes, encapsulation, inheritance, and polymorphism. A class is usually described as the template or blueprint from which the object is actually made. When you create an object from a class, you are said to have created an instance of the class. For example, all forms in Visual Basic are instances of the form class and an individual form in the application is actually a class you can use to create new forms. The controls on the toolbox represent individual classes, but an individual control on a form does not. Encapsulation is a key concept in working with objects; it is combining the data and behavior in one package. Encapsulation is the way to give an object its "black box"-like behavior and is fully supported by Visual Basic. Inheritance is the ability to make classes that descend from other classes. The purpose of

inheritance is to make it easier to build code for specialized tasks. Visual Basic does not support inheritance for creating new subclasses. Subclasses will usually inherit the same methods as the parent classes. OOP allows people to define a new method in a subclass but give it the same name (so-called overriding). A true OOP language like C++ allows people to go beyond simply overriding a method into what is usually called polymorphism. The idea behind polymorphism is that while the message may be the same, the object determines how to respond. Polymorphism can apply to any method that is inherited from a base class. Visual Basic does not support polymorphism in any form for the objects people create.

The Objective of the Thesis

Motivated by the fact that formal methods are gaining popularity in industry and business applications, the author will use formal methods and the object-Z specification language to design and write a specification for the University Sports Center's Information Management System. The author will use Microsoft Visual Basic 4.0 to implement the corresponding software based on the specification written down. The completed Windows-style software will provide a friendly user interface for both professional and non-professional users.

The case above is totally new, and the completed software is believed to be applicable in the real world. The author received good training in software design and Visual Basic programming through this work.

The Organization of the Thesis

This thesis presents the introduction and literature reviews of the research, the detailed steps of the specification analysis, and how it leads to the system implementation by using Microsoft Visual Basic 4.0. The detailed description of Z notations and schemas in our specification are presented in Appendix A and Appendix B, respectively.

Chapter 1, which is the current chapter, is an introduction that gives an overview of formal methods, Z and Object-Z specification languages, and Microsoft Visual Basic 4.0 and its relation to Graphical User Interface (GUI) and Object-Oriented Programming (OOP).

Chapter 2 includes the literature reviews for this research. We present a few applications of formal methods in industry and business, such as formal methods and CICS, and formal methods and telephony, etc.

Chapter 3 is the requirement and specification of the University Sports Center's Information Management System. In this chapter, we state the requirements and present and analyze the specification of the University Sports Center's Information Management System.

Chapter 4 is the system implementation of the University Sports Center's Information Management System. We will see how many database tables there are in the system, what the system screens look like, and how the developed system satisfies the specification stated in Chapter 3.

Chapter 5 is the summary and conclusion of the thesis. It is followed by references, Appendix A, and Appendix B. Appendix A gives a detailed description of Z notations. Appendix B gives schemas in our specification.

CHAPTER II

LITERATURE REVIEW

As mentioned earlier, much research has been focused on using formal methods to design a computer application system, due to the fact that formal methods are very useful design approaches in designing an application program. Using formal methods to design application software is very helpful in reducing the risks of misunderstanding, especially when a group of people collaborate on a large project. It is also helpful in hardware design.

In the following sections, we name a few applications of formal methods and Z /Object-Z specification languages.

Formal Methods and CICS

CICS (Customer Information Control System) is an IBM family of software products which is used by businesses world wide to manage and process their business information. CICS comprises a monitor system providing generic facilities for online transaction processing, which can be tailored to meet customers' exact business requirements by means of an application programming interface (API) used to invoke CICS services from within the customer's own business applications.

The application of mathematical methods to the development of CICS began in 1982 with a collaboration between Hursley and the Programming Research Group at Oxford University on the use of the Z notation for specification [Collins *et al.*, 1989]. Initially, Z was used in the CICS code restructuring initiative for version 3.1.1 and since then it has been used for the specification of the most new components of CICS as well as parts of the API. The CICS restructuring initiative is aimed at providing future CICS implementations with a frame work for modular design. Within this new structure, code modules are grouped according to the data upon which the code operates. Such a group of code modules is given the name *domain*. Z is well suited to the specification of CICS domains, and consequently a considerable degree of success has been achieved with its use for specification by CICS designers. The use of Z results in a tangible increase in the quality of the CICS product while maintaining the overall programmer productivity. Although extra work is involved in writing the formal specifications and holding inspections, this additional effort is offset by a reduction in the amount of rework required during the subsequent design and coding stages. The majority of the actual savings which have been realized so far through the use of formal methods have resulted from a reduction in service costs after the product's release.

Formal Methods and Computer Network Protocols

Dahai Li and T.S.E. Maibaum applied formal methods to the specification and verification of computer network protocols. In order to build a large and complex computerized system, adequate specifications of the requirements and implementations

are essential to ensure the quality of the system. Accurate but flexible specifications in the design of protocols in distributed computer systems are needed, because the systems are usually large in size and involve some quite sophisticated protocol algorithms. These protocols can provide the necessary service to their users. Li and Maibaum presented a logic formalism for the formal specification and verification of protocol problems. A token passing protocol on a ring network was developed, following the top-down step-wise refinement methodology, to demonstrate the practicality of formal methods in the process of protocol design [Li and Maibaum, 1988]. They benefited very much from one of the advantages of using formal methods: the elimination of ambiguities in system specification.

Formal Methods and Telephony

As mentioned before, Z is one of the specification languages in formal methods that has been widely used recently. Peter Mataga and Pamela Zave used Z to specify telephone features, specifically the call processing and subscriber database aspects. The specification was decomposed into two major parts: the telephone interface specification and the connections specification. The telephone interface part of the specification described the way in which features were invoked by the user via the telephone. The connection specification described the consequences of call processing and the maintenance of subscriber database information [Mataga and Zave, 1995]. Mataga and Zave sketched enough of the structure to allow a discussion of their experiences with the multiparadigm specification technique with feature specification and especially with Z.

They proved that it was possible to write a compact specification of a group of realistic subscriber telephone features in a framework which was suitable for extension. The use of Z can meet many of the goals for the specification, and it is significantly more concise than the corresponding English language feature specification documents. Because Z provides the abstraction, the feature descriptions are better suited to the requirements phase of a software development process.

Formal Methods and Computer Hardware Design

Formal methods are used not only in software design, but also in some hardware design. C.J. Dodge and P.E. Undrill used Z to create an application in digital hardware design. They presented an experiment using Z in the development of a hardware system. Giving part of the specification as an example, they showed how it was used to verify certain aspects of the hardware's behavior. They also showed "the refinement process from the abstract specification statements into realized hardware and programmable logic [Dodge and Undrill, 1996]." Dodge and Undrill concluded with a discussion of the merits of an abstract specification using Z for hardware design. They used Z in the FFT accelerator design in their project. The FFT accelerator is a combination of a dedicated Digital Signal Processing (DSP) based FFT processor with a multitransputer system. In Dodge and Undrill's project, they proved that Z provided a precise method of stating the required operations of the system, which, as they said, "dramatically helped systematic thought concerning the decomposition of the accelerator into more manageable blocks and subsequently the concise expression of the operation of each block [Dodge and

Undrill, 1996]." They showed that Z enabled a useful level of abstraction during design. Because the designer had a good grasp of the total system stage by stage, it was very straightforward for the designers to make decisions about the implementation during the refinement of the specification, and it was also straightforward for them to understand the implications of each decision. Using the Z specification language for hardware design also helped them in the derivation of control signals, and provided "a precise set of documentation notes which had been used as a reference while building the board and forming the basis of the accelerator documentation [Dodge and Undrill, 1996]." In Dodge and Undrill's project, the accelerator was simply expressed as a set of states or functions bringing about state changes. By using Z, the notation in the project was well suited to the hierarchical decomposition of both structure and behavior. By using Z, it allowed details to be put at various levels of abstraction and permitted logical reasoning about the behavior of signals.

Some Specification Languages in Formal Methods

Because formal methods are useful in computer design, people have developed several kinds of specification languages in formal methods. In K. Lano and H. Haughton's research, they described development techniques for B Abstract Machine Notation (AMN), including the "formalization of requirements, specification construction, design and implementation [Lano and Haughton, 1995]." The B AMN specification language is used widely in formal methods. B AMN has robust, commercially available tool support for the development life-cycle from specification to code generation. B AMN inherits the

advantages of its predecessor, the Z specification language. Today, B AMN is also widely used in both industry and academia as a software development tool.

LOTOS is a process-based formal specification language developed by the ISO for the formal description of OSI (Open System Interconnection) service and protocol standards. It is a type of formal description language which A.J. Tocher used to describe a communication standard. In the communication service which was developed by Tocher, users may establish, maintain, and relinquish communication connections between each other. Connections are established between named addresses. The messages are communicated at the endpoints. As Tocher writes, "The presentation of a relatively complete specification in LOTOS of a real distributed communications service has shown that the language is sufficiently expressive to address problems [Tocher, 1988]." This practice shows that using formal specification languages to design practical applications can represent the need for the development and expression of industrial standards. There are also some benefits to be gained from the use of formal methods in the development and expression of industrial standards.

There are several other specification languages, such as CCS, CSP, and VDM mentioned by the International Electrotechnical Commission. CCS stands for Calculus of Communicating Systems. It is very effective in expressing the composition of subsystems into a global system, but it prohibits any intermediate development steps [Bruns and Anderson, 1995]. CSP stands for Communicating Sequential Processes. It is a process-based formalism for the description of concurrent systems, and possesses a rich set of mathematical laws for their analysis[Hinchey and Bowen, 1995]. VDM is the Vienna Development Method. Its specification language (VDM-SL) is currently being

standardized under the auspices of the ISO and the British Standards Institution. It is model-oriented and has the large user community. A number of tools have been developed to support VDM-SL [Hinchey and Bowen, 1995].

The author chose the Z specification language (strictly speaking, Object-Z) because it is popular with governments, academics, and parts of industry, especially those developing critical systems where the reduction of errors and quality of software is extremely important. The other benefits people can gain from using Z have been stated in chapter I and the following part of chapter II.

Z and Object-Z

Because an object-oriented approach to software development is becoming popular, some people have started to use Z to create object-oriented applications. K. Periyasanmy and C. Mathew did a mapping from a functional specification to an object-oriented specification. They described a methodology to transform the Z specification into an Object-Z specification [Periyasamy and Mathew, 1996]. The implementation derived from the Object-Z specification is easy to maintain , enhance, and reuse.

MooZ (Modular Object-Oriented Z) is also an object-oriented extension of the Z specification language aimed at the specification of large software systems. By using MooZ, P. Duarte de Lima Machado and S.L. Meira presented an object-oriented formal specification of artificial neural networks used in the development of a simulation environment called EASY (An [E]nvironment for [A]rtificial Neural [SY]stems Simulation) [Machado *et al.*, 1994]. We know that neural networks are fault tolerant

because the failure of one or more neurons or connections may not result in a loss of knowledge. However, software implementations of neural networks must be reliable, because bugs may change, as in the behavior of a learning algorithm, compromising the final results. Mooz made possible a precise and unambiguous description of artificial neural network properties. Concerning the theory of neural networks so far, a great number of researchers around the world have been extending or creating new concepts and paradigms. Very little effort has been done to establish standards or to formalize concepts. The formal specification presented in their work represents a formalization of some of the basic neural network concepts. In Machado's work, the inclusion of new concepts and paradigms was done via the reuse of abstract class definitions in a formal way. This avoids the respecification of all basic features of neural networks and, indeed, reduces the development effort.

CHAPTER III

SPECIFICATION OF UNIVERSITY SPORTS CENTER'S INFORMATION MANAGEMENT SYSTEM

Z: basics and notations

Z is a set-theoretic specification language. Specifications in Z describe sets, and their constructs have their meaning in operations on sets. It is natural to know what are sets theory, what essential properties they have, and what operations can be performed on them. Mathematicians have established the foundations of set theory as an axiomatic theory of first-order logic. The following are basics of and notations in Z that will be used in our specification. A detailed description of Z notations is presented in Appendix A.

Three built-in sets

Z recognizes three built-in sets: the natural numbers **N**, the integers **Z** (whole numbers which range from minus infinity to plus infinity), and the natural numbers excluding 0 which are defined as:

```
| number: P N
|----------------------
| number = N \ {0}
```

All the standard operators on sets are defined in Z, such as difference, union, and intersection.

Schema

Schema is one of the principal features of Z . A schema has two sections, the part above the middle line in the figure is known as *signature* and the part below the middle line is known as *predicate*.

```
----SchemaName-----------------
|  signatures
|--------------------
|  predicates
---------------------------------------
```

For example, here is a schema called PieceOwned

```
---PieceOwned----------------------------
| piece ? : PIECE
|-------------------------
| piece ? ∈ available ∪ checkedout
-----------------------------------------------
```

in our specification, *piece* is the signature of the schema PieceOwned, and *piece ? ∈ available ∪ checkedout* is the predicate of the schema. There are many schemas in this specification (see Appendix B). In Object-Z, the schema is called the member function of a class. The declarative information in a schema is captured in its signature. This records the names of the schema's components or local variables, their types, and the given-set

names assumed by the schema. Signatures are finite objects, and are suitable for mechanical representation and manipulation. A schema contains more information than just the declarations. The axiom part of the schema can describe a relationship among the variables. The relationships among the variables in a schema are written as predicates. These predicates must always be true in every state of the system.

Basic Type Sets

Z specification provides a number of facilities which enable system specification to be built up easily. The first facility allows the specifier to declare the sets which are basic types in a specification. Those sets which are assumed to exist can be used in specifications and do not require any further definition. These sets are declared by enclosing them in square brackets. For example, in the specification of the University Sports Center's Information Management System, in order to describe the information of users in the system, we introduced the sets ID, NAME, ADDRESS, PHONE#, and TYPE to represent user's id, user's name, user's address, user's phone number, and user's type (staff or student), so we have:

[ID, NAME, ADDRESS, PHONE#, TYPE] .

In order to describe the information of equipment in the system, we introduced the sets ID, NAME, PLACE, PIECE, and SUBJECT to represent equipment's id, equipment's name, equipment's place, equipment pieces, and equipment's subject (indoor or outdoor), so we have:

[ID, NAME, PLACE, PIECE, SUBJECT].

Relations and Functions

Relations are also used in Z specifications. A relation between two types T_1 and T_2 is written in the signature part of a Z schema as

$$T_1 \leftrightarrow T_2.$$

Functions are a special type of relation; the set and relational operators can be used with Z objects that are described as functions. In the specification of the University Sports Center's Information Management System, we have partial functions and total functions for which the corresponding symbols are $\nrightarrow$ and $\rightarrow$.

Object-Z: An Extension of Z

Object-Z is an object oriented approach to Z. It offers plenty of help with structure as objects which can be used to split up a system's state space and inheritance to build up complexity. Object-Z makes it possible to write clear, abstract specifications of classes and inheritance hierachies that are readily comprehensible.

The major extension in Object-Z is the class schema which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are object references, i.e., identifiers which reference objects of the class. By declaring variables of class types within a class, objects which refer to and use other objects may be specified.

Class schema in Object-Z

An Object-Z class schema, often referred to simply as a class, is represented syntactically as a named box possibly with generic parameters. In this box there may be local type and constant definitions, at most one state schema and associated initial state schema, and zero or more operations. In addition to these explicit definitions, a class may inherit the definition of one or more other classes. The basic structure of a class is as the following.

```
-----ClassName--------------------------
| inherited class designators
| local type and constant definitions
| state schema
| initial state schema
| operations
-----------------------------------------
```

The various components of the class definition are described below.

Inherited class designators: When a class is inherited, its local types and constants are implicitly available in the inheriting class. Any types or constants with the same name occurring in both the inherited and the inheriting class are semantically identified and hence must have compatible definitions. The inherited class' state schema and initial state schema are implicitly conjoined with those declared explicitly in the inheriting class. The inherited class' operations are implicitly available except where the name of an operation in the inheriting class is the same as that of an operation of the

inherited class.

Local type and constants: The local type and constant definitions of a class have the same syntax as global type and constant definitions in Z. Their scope, however, is limited to the class in which they are declared. A constant is associated with a fixed value which cannot be changed by any operation of the class. They may be different for each instance of the class, but do not change during the lifetime of an object.

State schema: The unnamed schema defines the state and any state invariant. The state invariant is assumed to hold both before and after any operation.

Initial state: The initial state schema is named with the keyword INIT and has only a predicate part. It implicitly includes both the declarations and the predicates of the state schema. It specifies the initial state of the object.

Operations: The operations are defined either as operation schemas or operation expressions. They are interpreted in the class' local environment enriched with the declarations and predicates of the state schema. An operation schema extends the notion of a schema in Z by adding to it a delta-list. The Δ list gives all those state components that can change during the operation; all the others remain the same.

Requirement Analysis

Requirement analysis is the base for the system design. The requirement of the system developed is as the following:

- The university sports center has two types of users. Staff are responsible for recording transactions, retrieving, and lending sports equipment, and ordinary users include enrolled students and staff who can borrow sports equipment.
- When a user borrows sports equipment, the university sports center needs to record his or her ID number, name, address, and phone number, so that appropriate action may be taken if the sports equipment is not returned on time.
- Sports equipment is classified by ID number, number of pieces available, location, and subject, such as indoor sports equipment or outdoor sports equipment. The equipment may have several pieces available or may have only one available.

The developed system must satisfy the following requirements:

- Each piece of equipment can only have one status, either it is available for checkout or it has been already checked out .
- A user can only borrow up to the maximum number of equipment pieces at one time.

There are several actions which are transactions conducted by the university sports center.

- A user borrows a piece of equipment.
- A user returns a piece of equipment.

- Staff adds a piece of equipment to the university sports center.
- Staff removes a piece of equipment from the university sports center.
- Staff can get a list of equipment currently checked out by a particular user.
- Staff can find out who last checked out a particular piece of equipment.

The various transactions included in the system can be described by the following diagram.

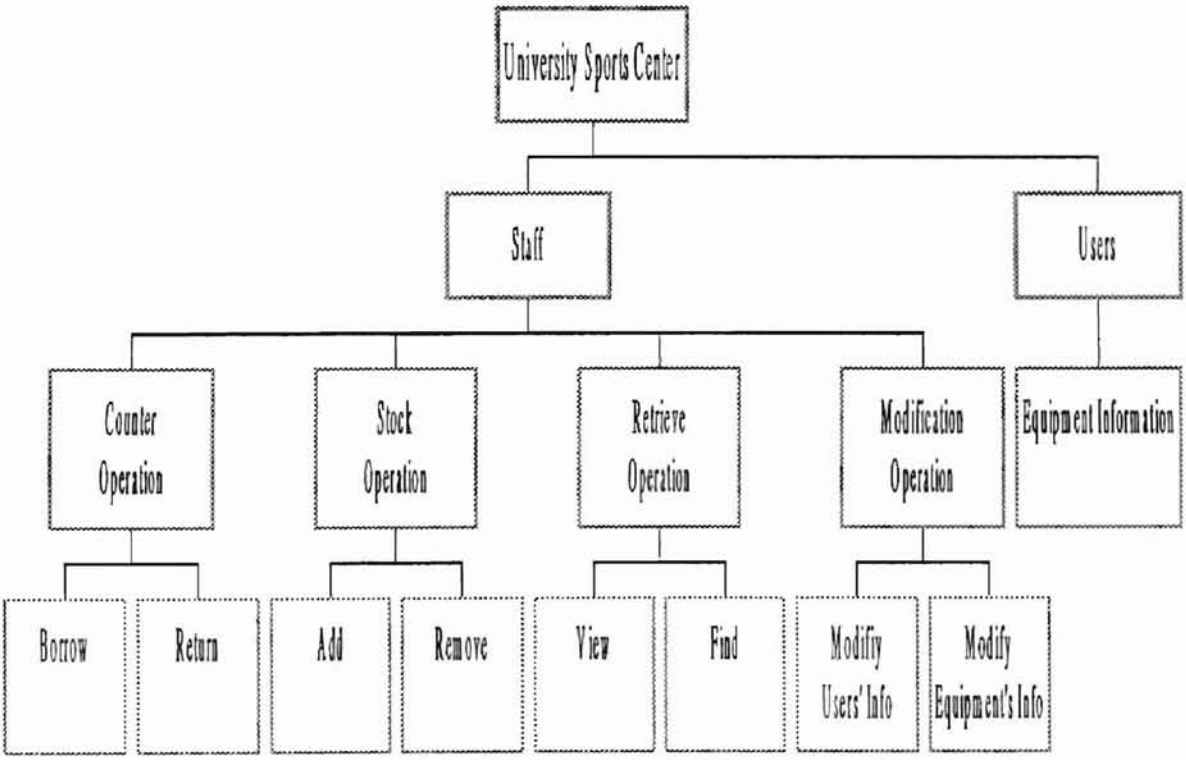

Figure 2. Diagram of various transactions in the system

Specification Analysis

Based on the requirement stated above, the formal specification of the University Sports Center's Information Management System is created. In the procedure, the corresponding mathematical statements which describe a view of the specification are developed.

We use Object-Z to write an object-oriented specification for the University Sports Center's Information Management System. In this specification, there are three major classes: UserClass, EquipmentClass and SportsCenterClass. UserClass includes most operations which have relation with users of the university sports center, such as searching the information about a user of the university sports center. EquipmentClass contains most operations which are performed to the equipment of the university sports center, such as stock operation which deals with adding and removing actions. SportsCenterClass inherits properties from UserClass and EquipmentClass and implements counter operations, that is, users of the university sports center can borrow or return a piece of equipment, but these operations can only be handled by the staff.

Abstract Classes

In the University Sports Center's Information Management System, there are two types of basic data that need to be manipulated: USER and EQUIPMENT, so there are two abstract classes in the specification to describe these two types:

---------- USER ----------
| id: ID
| name: NAME
| address: ADDRESS
| phone#: PHONE#
| type: TYPE
---

and

---- EQUIPMENT ----------
| id: ID
| name: NAME
| place: PLACE
| piece: PIECE
| subject: SUBJECT

Classes

In the specification of University Sports Center's Information Management System, there are three classes: EquipmentClass, UserClass and SportsCenterClass.

EquipmentClass:

State Schema

In the EquipmentClass, there are seven declarations.

| instanceof: PIECE ⇸ EQUIPMENT
| belong : ID → EQUIPMENT
| about: EQUIPMENT ⇸ SUBJECT
| available: **F** PIECE
| checkedout: **F** PIECE
| stock: **P** PIECE
| equipment-info: ID → EQUIPMENT
---

where *instanceof* is a partial function mapping from PIECE to EQUIPMENT. It means which equipment a piece is an instance of. *instanceof* describes that a piece can only be

an instance of one equipment but one equipment may have several pieces. The domain of this function is the set of all pieces of equipment in the university sports center, and the range of this function is the set of all the equipment in the university sports center. Because each kind of equipment is distinguished by its ID number, *belong* is a total function mapping from ID to EQUIPMENT, it tells us the one to one correspondence between every ID number and each kind of equipment. *ID* is a set of equipment's IDs in the university sports center. *about* is a partial function mapping from EQUIPMENT to SUBJECT, it tells us whether equipment is indoor or outdoor. *available* and *checkedout* are sets of all finite subsets of PIECE of equipment, these subsets contain all the pieces of equipment which haven't been checked out or have been checked out respectively. The intersection of *checkedout* and *available* is the empty set, that is, $checkedout \cap available = \varnothing$. *stock* is the set of all subsets of PIECE of EQUIPMENT. The union of *checkedout* and *available* is the set *stock*, that is, $checkedout \cup available = stock$. *equipment-info* is a function mapping from equipment ID number to specific equipment. It is a total function which describes that each kind of equipment corresponds to one ID number and can only have one description. The range of this function is *stock*, so $ran\ equipment\text{-}info = stock$. The initial value of those variables are set in the initial state schema.

Initial State Schema

----INIT----------
| $checkedout \cap available = \varnothing$
| $checkedout \cup available = stock$

Operations

According to the requirements stated, we need to have the following operations in the EquipmentClass: *AddNewEquipment, AddNewPiece, RemoveEquipment, RemoveAPiece,* and *Search.*

* AddNewEquipment operation

```
-----AddNewEquipment----------------------------------------------------
| Δ (belong, instanceof, about, available, stock, equipment-info)
| equipment ?: EQUIPMENT
| id ? : ID
| piece ?: PIECE
| subject ?: SUBJECT
|-----------------------------------------------
| id ? ∉ dom belong
| equipment ? ∉ ran instanceof
| belong' = belong ∪ { id ? ↦ equipment ? }
| available'= available ∪ { piece ?}
| instanceof' = instanceof ∪ { piece ? ↦ equipment ? }
| about' = about ∪ { equipment ? ↦ subject ? }
| stock' = stock ∪ { piece ? }
| equipment-info' = equipment-info ∪ { id ? ↦ equipment ? }
| checkedout' = checkedout
-------------------------------------------------------------------------------
```

In this member function, the values of variables *belong, instanceof, about, available, stock and equipment-info* of EquipmentClass will be changed after the operation. In this operation, we have input variables: *equipment, id, piece, subject.* The *equipment* is a new type of equipment which is going to be added to the university sports center, so it is not in the range of function *instanceof* , *id* is the ID number of new type of equipment, it is not in the domain of function *belong.* To implement the adding operation, the pair (*id, equipment*) is added to the *belong* function, and the pair (*piece, equipment*) is added to

the *instanceof* function. *available* is equal to the union of pieces owned by the university sports center and the new added pieces. *stock* is equal to all the pieces of equipment in the university sports center including the newly added pieces. The corresponding information about the new type of equipment is also added to the *equipment-info* function. The variable *checkedout* is unchanged. In order to implement adding new equipment to the university sports center, we need to have two auxiliary operations *NotNewEquipment* and *NewEquipment* in the class to justify whether the input equipment is already in the university sports center or not.

```
-----NotNewEquipment-------------------
| equipment ?: EQUIPMENT
|----------------------
| equipment ? ∈ ran instanceof
-----------------------------------------------

-----NewEquipment-----------------------
| equipment ?: EQUIPMENT
|------------------------
| equipment ? ∉  ran instanceof
-----------------------------------------------
```

Input *equipment* is the type of equipment which is going to be added to the university sports center. If it belongs to the range of the function instanceof, then it should be already in the university sports center, otherwise it is a new type for equipment of the university sports center.

* *AddNewPiece* operation

```
----AddNewPiece-------------------------------------------------
| Δ (available, instanceof, stock)
| piece ?: PIECE
| equipment ?: EQUIPMENT
|---------------------------------------
| piece ? ∉ available ∪ checkedout
| equipment ? ∈ ran instanceof
```

| available' = available ∪ { piece ? }
| stock' = stock ∪ { piece ? }
| instanceof' = instanceof ∪ { piece ? ↦ equipment ? }
| about' = about
| checkedout' = checkedout

In the member function EquipmentClass, we implement adding new pieces of equipment into the university sports center. We assume the equipment is contained in the university sports center, so we only need to add pieces to the certain equipment. In this case, the values of three variables *available, stock*, and *instanceof* have been changed. The two input variables are *piece* whose value is going to be added, and *equipment* which refers to the type of equipment that the pieces are going to be added to.

The operation of removing is split into two cases: one is removing all of the pieces of one type of equipment from the university sports center, and the other is removing a piece of equipment from the university sports center.

* *RemoveEquipment* operation

RemoveEquipment is a member function used to describe the action of removing a type of equipment from the university sports center.

----RemoveEquipment----------
| Δ (belong , instanceof)
| id ?: ID
| equipment ?: EQUIPMENT
|---
| id ? ∈ dom belong
| equipment ? ∈ ran instanceof
| belong' = belong \ { id ? ↦ equipment ?}
| instanceof' = instanceof \ {equipment.piece ↦ equipment ?}

In this state, the values of variables *belong* and *instanceof* are changed. Since each type equipment is distinguished by its ID number, we have the variables *id* and *equipment* as the input, The remove action is to remove the pair (*id, equipment*) from the *belong* function and in the meantime remove the pair (*piece, equipment*) from the *instanceof* function.

* *RemoveAPiece* operation

RemoveAPiece describes the action of removing a piece of equipment from the university sports center.

```
------RemoveAPiece----------------------------------------
| Δ ( available, stock )
| piece ?: PIECE
|-------------------------------------------------------
| piece ? ∈ available
| # (instanceof ⁻¹ (| { instanceof ( piece ?) }| ) ) >1
| available' = available \ { piece ? }
| stock' = stock \ {piece ? }
| checkedout' = checkedout
-----------------------------------------------------------------
```

In this state, we need to check whether the piece is in the *available* set or not. If it is in the *available* set and the number of pieces of this type of equipment is greater than one, then this piece is removed from the *available* and *stock* set. If the number of pieces of the equipment is equal to one, then this operation is the same as *RemoveEquipment*.

* *Search* operation

Another member function of EquipmentClass is *Search*. According to the input *id* value, we can get the information about this equipment.

```
-----Search----------------------------------------
| id ?: ID
| info !: equipment-info
|----------------------------------------
| id ? ∈ ran belong
| info ! = equipment-info(id)
-------------------------------------------------
```

UserClass

In the *UserClass,* there are four declarations: *student, staff, member* and *user-info. student* and *staff* are sets of all finite subsets of users. *member* is a function from user ID to USER and *user-info* is a function mapping from user ID to USER, used to get information about a specific user according to the user ID.

```
-----------------------------------
| student, staff: F USER
| member: ID → USER
| user-info: ID → USE
-----------------------------------
```

UnauthorizedRequestor operation

Only the staff of the university sports center are authorized to manipulate the operations in the information system, so we have a member function *UnauthorizedRequestor* to check whether the operator is staff or not.

```
----UnauthorizedRequestor------------
| requester ?: USER
| rep !: REPORT
| --------------------------------
| requester ? ∉ staff
| rep! = "Not authorized"
-------------------------------------------
```

UnregisteredUser

We have another member function called *UnregisteredUser* to check whether a user is registered to use the facility of the university sports center or not.

```
----Unregistered----------------------------
| person ?: USER
| rep !: Report
| ---------------------------
| person ? ∉ student ^ person ? ∉ staff
| rep! = "Not registered"
-------------------------------------------------
```

AddNewUser

In the UserClass, we need to implement adding a new user to the university sports center.

```
----AddNewUser-----------------------------------
| Δ ( member, staff, student)
| id ?: ID
| user ?: USER
|---------------------------------
| id ? ∉ dom member
| user ? ∉ student ∪ staff
| member' = member ∪ { id? ↦ user ? }
| user ?.type = staff
| staff' = staff ∪ { user ? }
| user ?.type = student
| student' = student ∪{ user? }
-----------------------------------------------------
```

A new user is not a current user of the university sports center. This means he or she is not in the domain of the function *member* and he is not in the union of the sets *student* and *staff*, but we need to add this new information into the university sports center. In this state, a pair (*id, user*) is added into the *member* function. If the user's type is staff then it will be added it into the *staff* set, otherwise it will be added into the *student* set.

RemoveUser

To implement removing a user from the university sports center, we have the following specification:

```
-----RemoveUser----------------------------------
| Δ ( member, user-info )
| id ?: ID
| user ?: USER
|-----------------------------------
| id ? ∈ dom member
| user ? ∈ ran member
| member' = member \ { id ? ↦ user ? }
| user-info' = user-info \ { id ? ↦ user ? }
-------------------------------------------------------
```

We input the ID number of the user who is going to be removed. ID is in the domain of the function *member*, and the corresponding user belongs to the range of the function *member*. So we delete (*id, user*) from the *member* function, and at the same time, we delete it from the *user-info* function. The information about the deleted user is not included in the university sports center anymore.

Search operation

The member function *Search* in the University Sports Center's Information Management System describes the function of querying the information about a specific user. The user's ID number is input, and information about this user is output.

```
--- Search-------------------------------
| id ?: ID
| info !: user-info
|-----------------------------------
| id ? ∈ dom user-info
| info ! = user-info (id?)
-------------------------------------------
```

SportsCenterClass

In SportsCenterClass, we implement checking out equipment and returning equipment operations. In this class, we declare a constant

| MaxPieceAllowed: N

MaxPieceAllowed is a local constant in the SportsCenterClass, which means the number of pieces borrowed by a user cannot exceed the limit set by MaxPieceAllowed.

State Schema

We have the following state schema:

| borrowedby, previouslyborrowedby: PIECE ⇸ USER
| EC: EquipmentClass
| UC: UserClass

--------INIT----------

| EC.available ∩ EC.checkedout = ∅
| UC.student ∩ UC.staff = ∅
| dom borrowedby ⊆ EC.checkedout
| ran borrowedby ⊆ UC.student ∪ UC.staff
| dom previouslyborrowedby ⊆ UC.available ∪ UC.checkedout
| ran previouslyborrowedby ⊆ UC.student ∪ UC.staff
| ∀ user: USER | user ∈ UC.student ∪ UC.staff
| • # borrowedby^{-1}(| { user } |) ≤ maxPieceAllowed

---

where *Borrowedby* and *Previouslyborrowedby* are two partial functions mapping from PIECE to USER. They tell us who is the current borrower of a piece of the equipment and who previously borrowed this piece of equipment. A piece of equipment can be borrowed by only one user, but a user can borrow several pieces at the same time. In this class we

declare two variables. One is EC, declared as EquipmentClass type, and the other one is UC, declared as UserClass type. In this case, we can use all variables in class EquipmentClass and class UserClass. In the initial state schema we assume the union of the sets *available* and *checkedout* cannot be an empty set. The union of sets *student* and *staff* cannot be an empty set either. The pieces which are borrowed by someone are in the *checkedout* set of EC class. The user who borrowed the pieces of equipment is in the union of sets *student* and *staff* of the UC class. The user who is borrowing the equipment cannot borrow more than a limited number of pieces.

There are two member functions in the SportsCenterClass, which are used to implement the counter operations of the university sports center.

CheckoutPiece operation

The member function *CheckoutPiece* describes the procedure of checking equipment out.

```
-----CheckoutPiece-----------------------------------------------
| Δ (borrowedby, EC.available, EC.checkedout)
| user ?: USER
| piece ?: PIECE
|----------------------------------
| user ? ∈ UC.student ∪ UC.staff
| piece ? ∈ EC.available
| # borrowedby^-1(| { user?} |) < maxPiecesAllowed
| EC.available' = EC.available \{ piece ? }
| EC.checkedout' = EC.checkedout ∪ { piece ? ↦ user? }
| borrowedby' = borrowedby ∪{ piece ? ↦ user? }
| previouslyborrowedby' = previouslyborrowedby
| UC.student' = UC.student
| UC.staff' = UC.staff
-----------------------------------------------------------------------
```

The check-out operation requires that a piece of equipment be available for checking out, and that users be eligible for borrowing. The number of pieces that the user has borrowed must not exceed maxPiecesAllowed. After the transaction, the piece that has been borrowed is removed from the set *available*, and added to the *checkedout* set. The pair (*piece, user*) is added to *borrowedby*. Other variables in the EC and UC have not been changed.

Return operation

To implement the *return* transaction, we have:

```
----Return-------------------------------------------------------------
| Δ (EC, borrowedby, previouslyborrowedby)
| piece?: PIECE
| user ?: USER
|--------------------------
| piece? ∈ EC.checkedout
| user ? ∈ ran borrowedby
| EC.available' = EC.available ∪ { piece? }
| borrowedby' = borrowedby \ { piece ? ↦ user? }
| EC.checkedout' = EC.checkedout \ { piece ? }
| UC.student' = UC.student
| UC.staff' = UC.staff
-----------------------------------------------------------------------
```

The piece of equipment must belong to the *checkedout* set. After the implementation of the return transaction, the returning piece is added back to the *available* set. The *borrowedby* information is changed by deleting the pair (*piece, user*). The returning piece is deleted from the *checkedout* set. Other variables are not changed in this state.

This specification presented above shows one possible way in which we can use the Object-Z specification language to write a specification to satisfy the requirement of the University Sports Center's Information Management System.

CHAPTER IV

SYSTEM IMPLEMENTATION

The software of University Sports Center's Information Management System is implemented by using Visual Basic 4.0. Programming under Visual Basic 4.0 can provide an advanced Windows interface. The University Sports Center's Information Management System is running on Windows 3.1. Windows is a very popular operating system, because the user interface is the same for all Windows applications, so the users who are going to operate this application of the University Sports Center's Information Management System would not be taught how to operate the system. They can minimize and maximize the windows and complete other conventional Windows operations.

General Design in Visual Basic

Visual Basic 4.0 is a tool to develop GUI-based applications efficiently. The objects placed on the windows are called *controls*. Controls in Visual Basic will recognize events like mouse clicks; how the objects respond to them depends on the code written on it. The core of a Visual Basic program is a set of independent pieces of code that are activated by, and respond to, the events they have been told to recognize [Cornell and Strain, 1995]. The programming code in Visual Basic that tells the program how to *respond* to events like mouse clicks is called an *event procedure*. An event procedure is a body of

code that is only executed in response to an external event. Everything that is executable in a Visual Basic program is either in an event procedure or is used by an event procedure to help the procedure carry out its job.

To design the University Sports Center's Information Management System by using Visual Basic 4.0, we use the following steps:

- Customize the windows that the user sees.
- Decide what events the controls on the window should recognize.
- Write the event procedures for those events.

Database Design in Visual Basic 4.0

Based on its requirements, the University Sports Center's Information Management System is a database management system. The reason for using a database as the media to store the data is because it is easy to set up and manipulate and doesn't require massive resources. The database model used by Visual Basic 4.0 is a *relational database*. The *Data Manager* supplied by Visual Basic 4.0 is a handy tool for maintaining and creating databases. By using Data Manager we can implement the create, restructure , index, modify, copy, and query database tables. The Data Manager is the only way supplied with Visual Basic to build a database.

Visual Basic 4.0 can work with an existing database through the data control. By setting properties of the data control, the data control can be hooked to a specific database, then add controls to a form that will display the data. The data control itself

displays no data, it only conducts the flow of information back and forth between project and database. The ordinary Visual Basic controls are used to display the data. Controls that can work with the data control to access data are said to be *data-aware*, and the process of tying a data-aware control to a data control is called *binding* the data-aware control. Among Visual Basic Standard controls, the only intrinsic data-aware controls are text boxes, labels, check boxes, image controls, List boxes, Combo boxes, DBGrids, Masked edit, 3D Panel, and 3D check box. Data-aware controls must be on the same form as the data control, but they need not be visible in order to pick up the information. Once these controls pick up the information sent to them by the data control, the information will be stored as values of properties of the controls.

Implementation of the Management System

Structure

Design the functions of the University Sports Center's Information Management System. In order to satisfy the requirements, we have the following structure to describe the system:

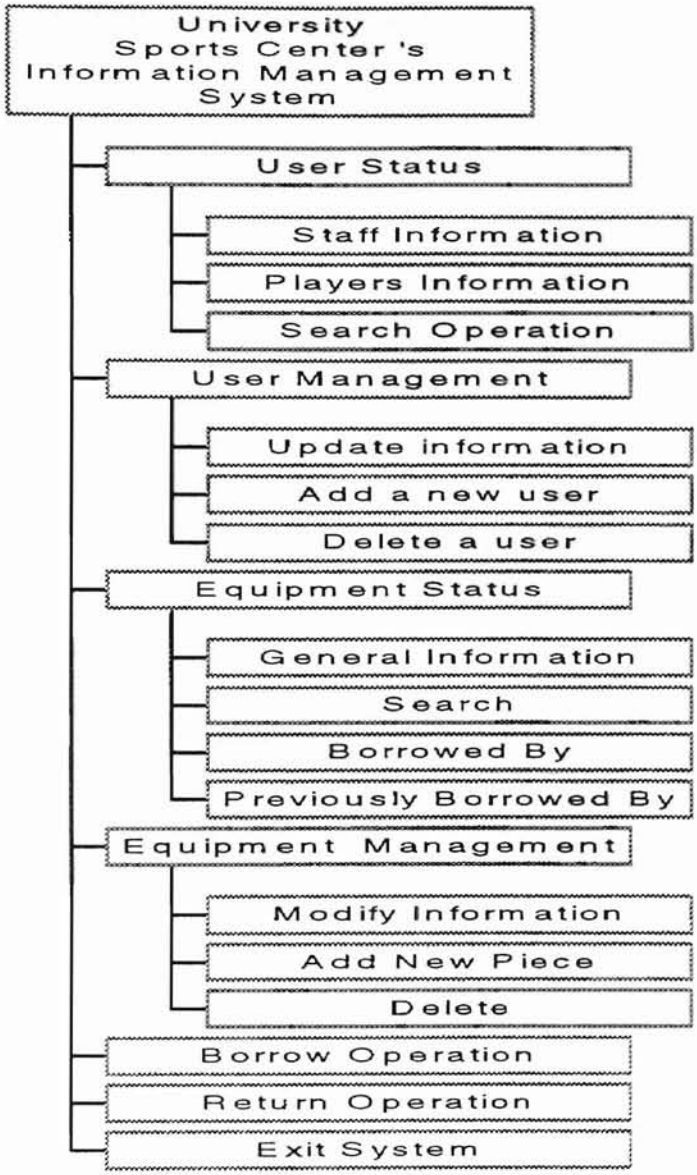

Figure 3. The structure of the University Sports Center's Information Management System

According to this structure, we design the main screen of the system to look like Figure 4.

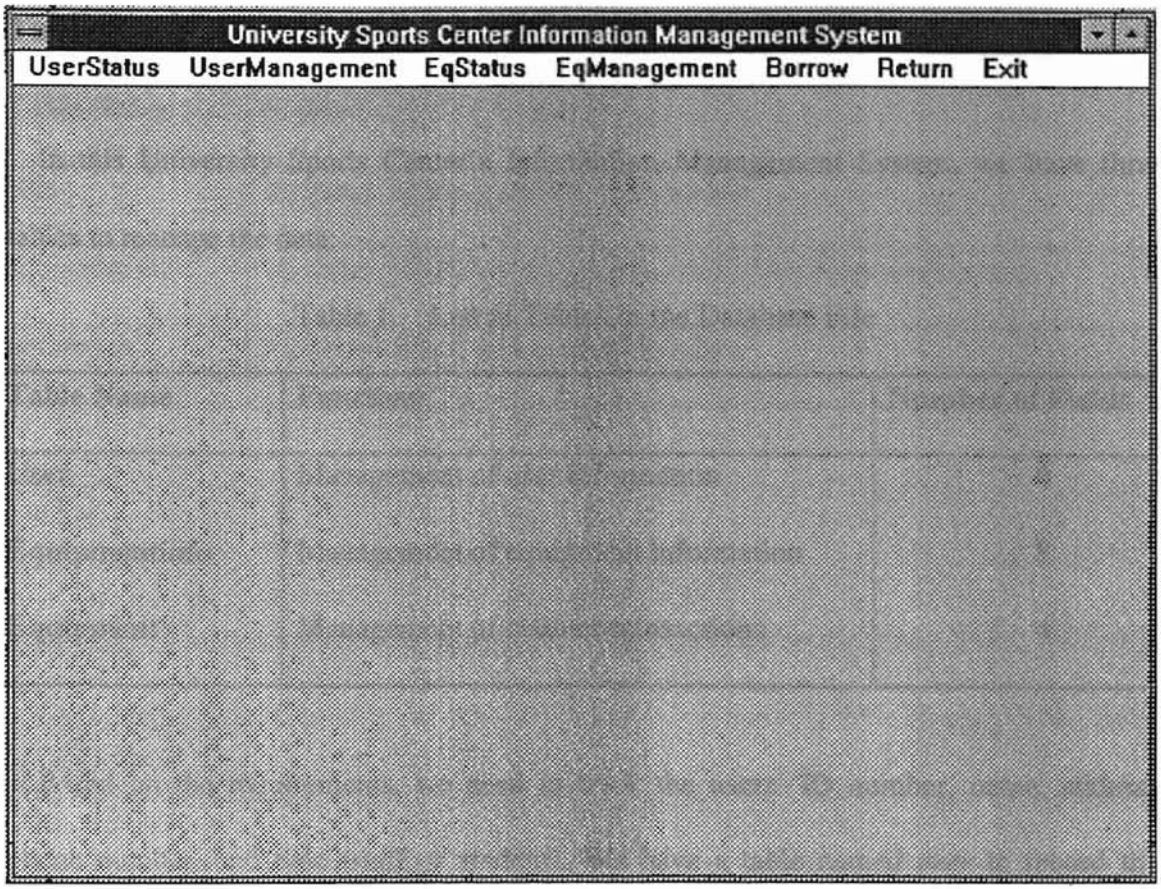

Figure 4. The main screen of the University Sports Center's Information Management System

The items in the menu and submenus on the main screen correspond to different functions in the specification of the University Sports Center's Information Management System. UserStatus, UserManagement, and their submenus implement functions of UserClass in the specification. EqStatus, EqManagement, and their submenus implement functions of EquipmentClass in the specification. Borrow and Return implement functions of SportsCenterClass.

Tables

In this University Sports Center's Information Management System, we have three tables to manage the data:

Table 1. List of Tables in the Database File

Table Name	Function	Number of Fields
User	Management of user information	6
Equipmentinfo	Management of equipment information	8
Equipment	Management of counter transactions	4

Based on the requirements, we need to track the users' ID number, name, address, phone number, and type (staff or student). We have a table named *user* to record this information about all users who are eligible to use the facilities in the university sports center. We also create a table named *equipmentinfo* to record information about all the equipment in the university sports center, such as equipment ID (because a type of equipment is distinguished by its ID number), equipment name, subject, location, how many pieces of this equipment that the university sports center owns, how many pieces are available, and how many pieces have been checked out. The counter transactions of the university sports center include borrowing and returning a piece of equipment. In order to record this information, we have the table *equipment* to record which type of equipment is currently borrowed, by whom, the borrowing date, and who previously borrowed this type of equipment .

Table 2. List of Fields in All Tables

Table Name	Fields
user	id, name, address, phone, type, memo
equipmentinfo	id, name, place, subject, pieces, available, borrowed, memo
equipment	id, borrowedby, preborrowedby, date

Index

In order to accelerate the searching speed, we create several indexes. Index *IDBorrow* is used to speed up searching for who is currently borrowing a type of equipment. Index *EquipId* is used to speed up searching for the information about a type of equipment according to its ID number. Index *EquipName* is used to speed up seeking the information about a type of equipment according to its name. *IndexId* and *MyIndex* are used to seek the information about a user according to his id number and name respectively.

Table 3. List of Indexes for All Tables

Index Name	Table Name	Field Name
IDBorrow	equipment	id
EquipId	equipmentinfo	id
EquipName	equipmentinfo	name
MyIndex	user	name
IndexId	user	id

These indexes are all in ascending order.

Forms

Visual Basic responds by displaying various windows on the desktop. One of the windows displayed is form. Forms will also respond to different events. There are five forms developed for the University Sports Center's Information Management System corresponding to the classes in the specification, and the events that occur in each form correspond to the operations in the class. The following table describes the functions of the different forms.

Table 4. List of all forms and their functions in the system

Name Form	File Name	Function
frmmain	THESIS.FRM	There is a menu in this form; each item in the main menu will conduct to a form.
frmUser	USER.FRM	In this form, the operations of userClass in the specification will be implemented, including browsing the information about all users, adding and deleting a user, and searching the information.
frmEquip	EQUIPMENT.FRM	In this form, the operations of

		equipmentClass in the specification will be implemented, including browsing the information about all equipment, adding, deleting equipment, and searching the information.
frmBorrow	BORROW.FRM	In this form, the borrow operation of SportsCenterClass in the specification will be implemented.
frmReturn	RETURN.FRM	In this form, the return operation of SportsCenterClass in the specification will be implemented.

In each form, there are several controls. We have text boxes, labels, DBGrids, data controls, and Combo list. At run time, the form provides the view to the user who is going to operate the system.

The user form looks like the following at run time when the user clicks the item *Add* in the UserManagement.

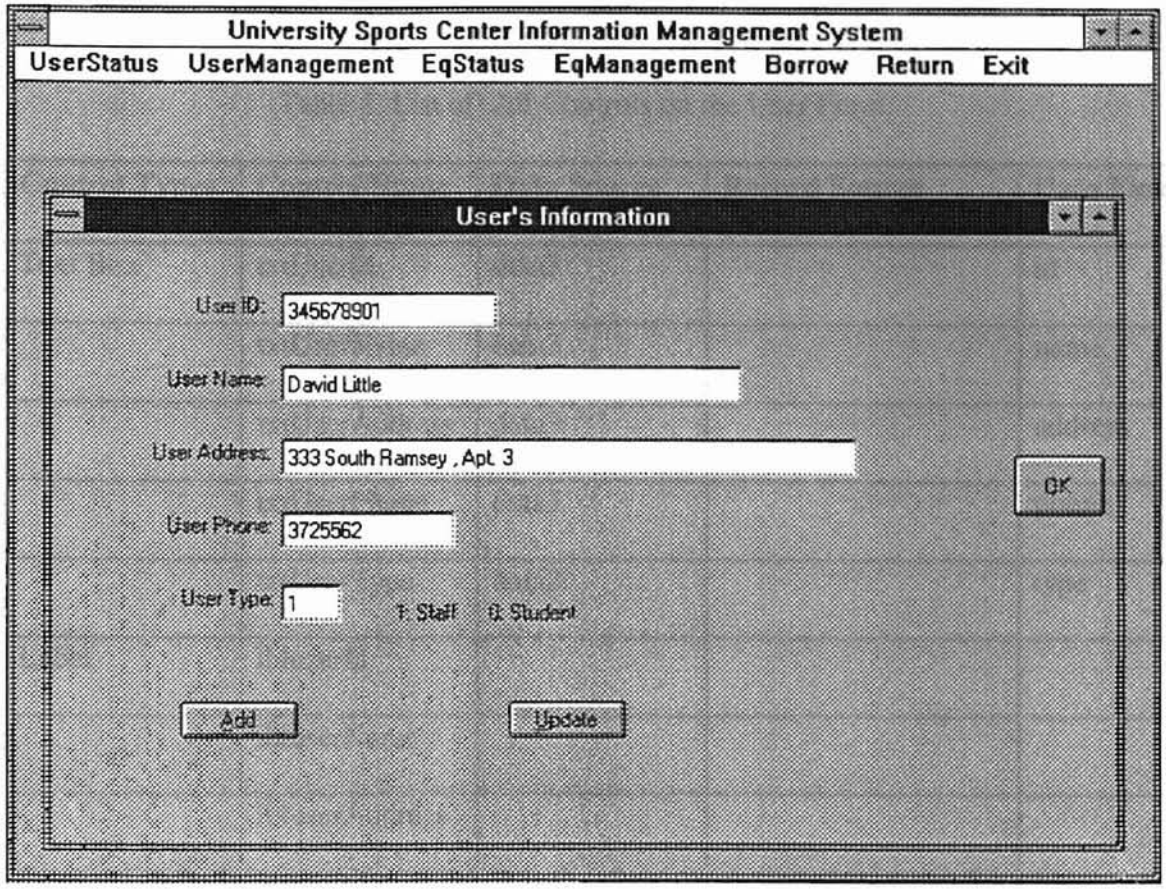

Figure 5. The view provided by the user form

Visual Basic 4.0 provides the facility (data control) to connect the controls with database table fields. Most controls on the forms are connected with a data control.

The controls on the user form are summarized in the following table:

Table 5. List of All Controls on the User Form

Control Types	Control Name	Data Source	Record Source	Data Field
Text Box	txtUserId	data3		id
	txtUserName	data3		name
	txtUserAddress	data3		address
	txtUserPhone	data3		phone
	txtUserType	data3		type
Label	lbluserId			
	lbluserName			
	lbluserAddress			
	lbluserPhone			
	lbluserType			
	lblBorrow			
DBGrid	DBGridBorrow	data2		
Data	Data1		user	
	Data2		equipment	
	Data3		user	
Button	cmdAdd			
	cmdDelete			
	cmdUpdate			

Among the three data controls on the user form, the Recordset type of data1 and data2 are *Dynaset*. The dynaset-type Recordset is an object of type Recordset object that can be used to manipulate data in an underlying table or tables. The dynaset-type Recordset is a dynamic set of records that can contain fields from one or more tables or queries in a database and may be changed. The dynaset-type Recordset is different from the snapshot-type Recordset because only a unique key for each record is brought into memory, instead of actual data. As a result, a dynaset is normally updated with changes made to the source data, while a snapshot is not. Like the table-type Recordset, a dynaset's current record is fetched only when its fields are referenced. The Recordset type of data3 is table. A table-type Recordset object is a representation in code of a base table that can be used to add, change, or delete records from a table. Only the current record is loaded into memory. A predefined index is used to determine the order of the records in the Recordset object.

When clicking the item *AddNew* in *EqManagement,* we get the following screen:

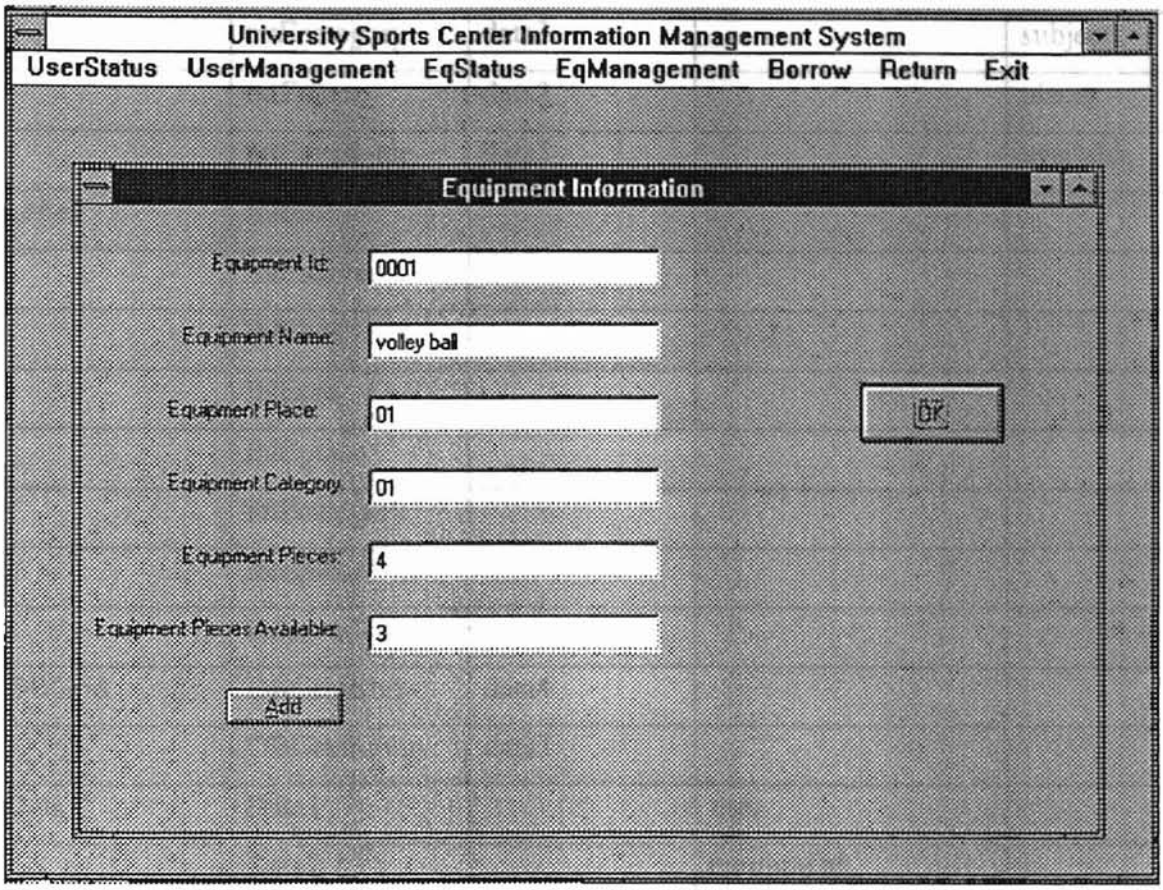

Figure 6. The view provided by the equipment form

There are two forms in this screen. One is *frmmain,* and the other one is *frmEquip* We cannot switch to the *frmmain* until the *frmEquip* form is closed, which is called modality. This property makes it safe to add a new type of equipment to the university sports center.

The controls on the *frmEquip* form are described in the following table.

Table 6. List of All Controls on the Equipment Form

Control Types	Control Name	Data Source	Record Source	Data Field
Text Box	txtEuserid	data3		id
	txtEname	data3		name
	txtEplace	data3		place

	txtEcategory	data3		subject
	txtEpiece	data3		pieces
	txtEavailable	data3		available
Label	lblEid			
	lblEname			
	lblEbor			
	lblEpre			
	lblEplace			
	lblEcategory			
	lblEpiece			
	lblEavailable			
DBGrid	DBGridBorr	data4		
	DBGridEquip	data3		
Data	Data1		user	
	Data2		equipment	
Button	cmdAdd			
	cmdDelete			
	cmdUpdate			

The Recordset type of data controls (data3 and data4) on this form are all *dynaset*.

The view of *frmBorrow* and *frmReturn* are the same, but each form implements different work. *frmBorrow* implements borrowing operations and *frmReturn* implements returning operations.

The view of *frmBorrow* and *frmReturn* looks like the following:

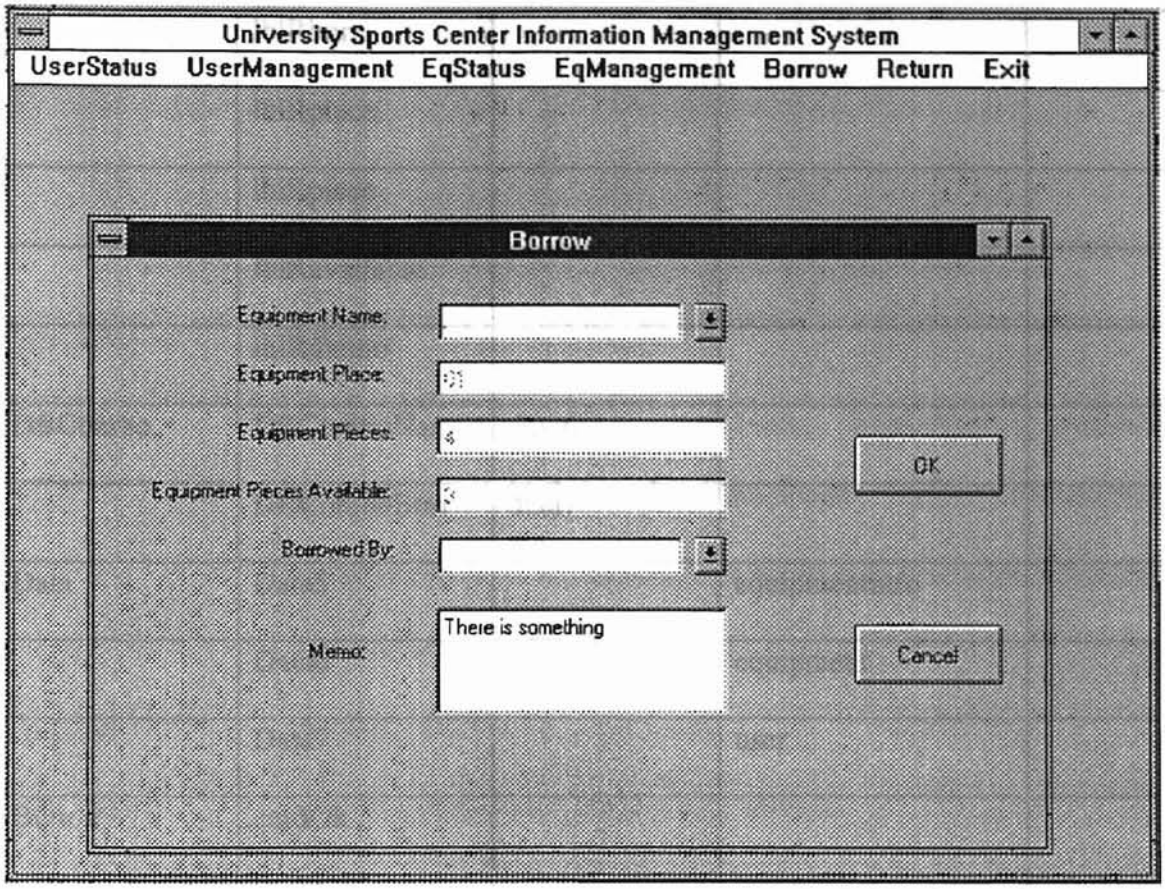

Figure 7. The view provided by the return form

Table 7 List of All Controls on the Return Form

Control Types	Control Name	Data Source	Record Source	Data Field
Text Box	txtBPlace	data5		place
	txtBPiece	data5		pieces
	txtBAvailable	data5		available
	txtBMemo	data5		memo
Label	lblRname			

	lblRborr			
	lblRplace			
	lblRpiece			
	lblRavailable			
	lblRmemo			
DBCombo	DBComboName	data6		
	DBComboBorr	data7		
Data	Data5		equipmentinfo	
	Data6		equipment	
	Data7		user	
Button	cmdOk			
	cmdCancel			

Among the three controls in the form *frmReturn*, the Recordset type of data5 and data7 is table, so we can use indexes for quick searching. The Recordset type of data6 is *dynaset*.

Event Procedures

After placing the objects in the form and setting their properties, the visual programming part of the job is completed. Visual Basic is an event-driven programming language. This means that code is executed as a response to an event. In order to

implement the functions of the University Sports Center's Information Management System, we generated appropriate codes and attached them to the objects and events.

Table 8. List of All Objects Which Have Event-procedures Embedded.

Form Name	Objects
frmmain	mnuStaff, mnuStudent, mnuFind, mnuUpdate, mnuAdd, mnuDelete, mnuGeneral, mnuSearch, mnuPreBorowedBy, mnuBorrowedBy, mnuModify, mnuAddNew, nuDeleteEquipment, mnuBorrow, mnuReturn, mnuExit.
frmuser	cmdAdd, cmdDelete, cmdUpdate.
frmEquip	cmdAdd, cmdDelete, cmdUpdate, cmdOk.
frmBorrow	DBComboName, DBComboBorr, cmdOk, cmdCancel.
frmReturn	DBComboName, DBComboBorr, cmdOk, cmdCancel.

Security and Error Detection

Security and Error detection is provided in the University Sports Center's Information Management System. We have several security and error detection methods, such as:

- When a user tries to operate stock and counter operations, he needs to input his user ID number to prove that he is staff, since only staff have this privilege.
- When a user is doing the searching operation, an error message will appear if the input value is not right or the input value is not included in the system.

- When a staff member is doing stock and counter operations, the input value will not be accepted if the input data does not satisfy a certain standard.
- When a user tries to borrow more than a maximum number pieces of equipment, the borrow action is not allowed.

CHAPTER V

SUMMARY AND CONCLUSIONS

In this thesis we used formal methods and the Object-Z specification language to design and write a specification for the University Sports Center's Information Management System. We used Visual Basic 4.0 to implement the software according to the specification.

Formal methods are very helpful to create high quality computer software and hardware. Formal methods are sets of mathematical notations and tools which can standardize the computer system's specification and design. Formal methods are one of a number of techniques that have been demonstrated that, when applied correctly, result in systems of the highest integrity.

The Object-Z specification language is one of the specification languages in formal methods. Object-Z is object extension to the Z specification language. It provides sufficient support for either encapsulation or inheritance. We used Object-Z specification language to write the formal specification for the University Sports Center's Information Management System.

Formal Specification is the use of notations derived from formal logical to describe: (1) assumptions about the world in which a system will operate, (2) requirements that the system is to achieve, and (3) a design to meet those requirements. We created such a specification for the system which we developed.

We used Visual Basic 4.0 to implement the software of University Sports Center's Information Management System because it can provide a Graphic User Interface (GUI). The GUI can usually provide users with several convenient data input and output formats, such as windows, menus, mice, image, and voice. With GUI, the system is more user-friendly, easy to learn, and easy to use. This is very important for non-professional users.

After the system was developed, we observed the advantages of formal methods. The system developed provides the user a friendly interface and is easy to use. The system is also easy to maintain.

The specification of University Sports Center's Information Management System presented here is only one possible route. It still needs improvement and optimization. One suggestion of future work is to divide the University Sports Center's Information Management System into two parts: one for the users of the university sports center to retrieve the information, and the other one for the staff of the university sports center for information retrieving and managing. This will make this system more convenient and efficient to use.

BIBLIOGRAPHY

Barden, R., Stepney, S., Cooper D. (1994). Z in practice Englewood Cliffs, NJ: Prentice-Hall International Limited.

Berg, H. E., Boebert, W. E., Franta W.R., Moher T. G. (1982) Formal Methods of Program Verification and Specification. Englewood Cliffs, NJ: Prentice-Hall International Limited.

Booch, G. (1991). Object Oriented Design with applications Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.

Bowen, J. & Gordon, M. (1995). A shallow embedding of Z in HOL. Information and Software Technology. 37(5), 269-276.

Bruno, G. (1995) Model-based Software Engineering. London, UK: Chapman & Hall.

Bruns, G. & Anderson, S. (1995). Gaining Assurance with Formal Methods. Application of Formal Methods (pp. 33-54). Englewood Cliffs, NJ: Prentice-Hall International Ltd.

Collins, B. P., Nicholls, J. E. and Sorensen, I. H. (1989) Introducing Formal Methods: The CICS Experience with Z. IBM Technical Report TR12.260, IBM UK Laboratories Ltd., Hursley Park, UK.

Davis, H. (1996) Visual Basic 4 Secrets. Chicago: IDG Books WorldWide, Inc.

Dodge, C. J., Undrill, P. E., Allen, A. R. & Ross, P. G. B. (1996). Application of Z in digital hardware design. IEEE Proceeding Computers and Digital Techniques, 143(1), 79-86.

Ford, N. J., Ford, J. M. (1993) Introducing formal methods a less mathematical approach.

New York: Ellis Horwood Limited.

Gurewich, N., Gurewich, O. (1995) Teach yourself Visual Basic 4 in 21 days. Indianapolis, Indiana: Sams Publishing.

Hinchey, M. G., Bowen, J. P. (1995). Applications of Formal Methods FAQ. Application of Formal Methods (pp. 1-5). Englewood Cliffs, NJ: Prentice Hall International Ltd.

Ince, D. C. (1992) An introduction to discrete mathematics, formal system specification, and Z. Oxford, UK: Oxford University Press, Inc.

Jones, C.B. (1986). Systematic Software Development Using VDM. Englewood Cliffs, NJ: Prentice-Hall International, 1986.

Lano, K., Haughton, H. (1995). Formal development in B Abstract Machine Notation. Information and Software Technology, 37(5), 303-316.

Li, D.H., Maibaum, T.S.E. (1988). Developing a high level specification formalism. Formal methods: Theory and Practice (pp. 53-102). Boca Raton, FL: CRC Press, Inc.

Machado, P. D. L., Meira, S. R. L., Gomes, H. M. (1994). EASY - an Environment for Artificial Neural Systems Simulation. In Proceedings of 4th Irish Neural Network Conference - INNC'94, Dublin, Ireland, September.

Mander, K. C., Polack, F. A. C. (1995). Rigorous specification using structured systems analysis and Z. Information and software technology , 37(5), 285-291.

Mataga, P., Zave, P. (1995). Using Z to specify telephone features. Information and Software Technology, 37(5), 277-283.

McKelvy, M. (1995). Using Visual Basic 4, The Fast and Easy Way to Learn. Boston: Que Corporation.

Periyasamy, K., Mathew, C. (1996). Mapping a functional specification to an Object-Oriented specification in software Re-engineering. CSC '96 (pp. 26-33), Philadephia.

Scharbach, P.N. (1988). Formal Methods: Theory and Practice. Formal methods: Theory and Practice (pp. 1-4), Boca Raton, FL: CRC Press, Inc.

Spivey, J.M. (1988). Understanding Z: a specification language and its formal semantics, Cambridge, UK: Press Syndicate of the University of Cambridge.

Tocher, A. J. (1988). LOTOS and the formal specification of communication standards: An example. Formal methods: Theory and Practice (pp. 5-51). Boca Raton, FL: CRC Press, Inc.

APPENDIXES

APPENDIX A

Z AND OBJECT-Z NOTATIONS USED IN THIS THESIS

$\mathbb{P}$	power set
$\varnothing$	empty set
#	cardinality of a set
$\pfun$	partial function
$\fun$	total function
$\bigcup$	distributed union
$\oplus$	relational override
$\mapsto$	maps to
$(\!\mid \mid\!)$	relational image
x?	input variable x
y!	output variable y
$\Delta(...)$	state variables modified by an operation

APENDIX B

SPECIFICATION OF THE UNIVERSITY SPORTS CENTER'S INFORMATION MANAGEMENT SYSTEM

[ID, NAME, ADDRESS, PHONE#, TYPE] ;

[ID, NAME, PLACE, PIECE, SUBJECT].

```
----- User----------------------
| id: ID
| name: NAME
| address: ADDRESS
| phone#: PHONE#
| type: TYPE
-----------------------------------

---- Equipment -----------------
| id: ID
| name: NAME
| place: PLACE
| piece: PIECE
| subject: SUBJECT
-----------------------------------

-----EquipmentClass-------------------------------------------------------------------------
|   ------------------------------------------
|   | instanceof: PIECE  ↦ EQUIPMENT
|   | belong : ID  →  EQUIPMENT
|   | about: EQUIPMENT  ↦ SUBJECT
|   | available: F  PIECE
|   | checkedout: F PIECE
|   | stock: P PIECE
|   | equipment-info: ID  → EQUIPMENT
|   ------------------------------------------
|
|   -----INIT---------------------------------
|   | checkedout ∩ available = ∅
|   | checkedout ∪ available = stock
|   ------------------------------------------
```

```
|
|  -----AddNewEquipment----------------------------------------------
|  | Δ (belong, instanceof, about, available, stock, equipment-info)
|  | equipment ?: EQUIPMENT
|  | id ? : ID
|  | piece ?: PIECE
|  | subject ?: SUBJECT
|  |--------------------------------------------
|  | id ? ∉ dom belong
|  | equipment ? ∉ ran instanceof
|  | belong' = belong ∪ { id ? ↦ equipment ? }
|  | available'= available ∪ { piece ?}
|  | instanceof' = instanceof ∪ { piece ? ↦ equipment ? }
|  | about' = about ∪ { equipment ? ↦ subject ? }
|  | stock' = stock ∪ { piece ? }
|  | equipment-info' = equipment-info ∪ { id ? ↦ equipment ? }
|  | checkedout' = checkedout
|  ---------------------------------------------------------------------------
|
|  -----NotNewEquipment---------------
|  | equipment ?: EQUIPMENT
|  |----------------------
|  | equipment ? ∈ ran instanceof
|  ----------------------------------------
|
|  ---- NewEquipment-------------------
|  | equipment ?: EQUIPMENT
|  |-------------------------
|  | equipment ? ∉  ran instanceof
|  ----------------------------------------
|
|
|  ----AddNewPiece-----------------------------------------------
|  | Δ (available, instanceof, stock)
|  | piece ?: PIECE
|  | equipment ?: EQUIPMENT
|  |---------------------------
|  | piece ? ∉ available ∪ checkedout
|  | equipment ? ∈ ran instanceof
|  | available' = available ∪ { piece ? }
|  | stock' = stock ∪ { piece ? }
|  | instanceof' = instanceof ∪ { piece ? ↦ equipment ? }
|  | about' = about
|  | checkedout' = checkedout
```

| ----------

| -----PieceCheckedOut----------

| | piece ?: PIECE

| |----------

| | piece ? ∈ checkedout

| ----------

| -----PieceAvailable----------

| | piece ?: PIECE

| |----------

| | piece ? ∈ available

| ----------

| ---------- RemoveEquipment----------

| | Δ(belong , instanceof)

| | id ?: ID

| | equipment ?: EQUIPMENT

| |----------

| | id ? ∈ dom belong

| | equipment ? ∈ ran instanceof

| | belong' = belong \ { id ? ↦ equipment }

| | instanceof' = instanceof \ {equipment.piece ↦ equipment }

| ----------

| -----RemoveAPiece----------

| | Δ (available, stock)

| | piece ?: PIECE

| |----------

| | piece ? ∈ available

| | # (instanceof $^{-1}$ (| { instanceof (piece ?) }|)) >1

| | stock' = stock \ {piece ? }

| | available' = available \ {piece ?}

| | checkedout' = checkedout

| ----------

| -----Search----------

| | id ?: ID

| | info !: equipment-info

| |----------

| | id ? ∈ ran belong

| | info ! = equipment-info(id)

| ----------

---------- UserClass----------

| student, staff: F USER
| member: ID → USER
| user-info: ID → USER

-----UnauthorizedRequestor----------

| requestor ?: USER
| rep !: REPORT
| requestor ? ∉ staff
| rep! = "Not authorized"

----------Unregistered----------

| person ?: USER
| rep !: REPORT
| person ? ∉ student ^ person ? ∉ staff
| rep! = "Not registered"

------AddNewUser----------

| Δ (member, staff, student)
| id ?: ID
| user ?: USER
| id ? ∉ dom member
| user ? ∉ student ∪ staff
| member' = member ∪ { id? ↦ user ? }
| user ?.type = staff
| staff' =staff ∪ { user ? }
| user ?.type = student
| student' = student ∪{ user? }

-----RemoveUser----------

| Δ (member, user-info)
| id ?: ID
| user ?: USER
| id ? ∈ dom member
| user ? ∈ ran member

| | member' = member \ { id ? ↦ user ? }
| | user-info' = user-info \ { id ? ↦ user ? }
| ---
|
| ------ Search----------
| | id ?: ID
| | info !: user-info
| |----------
| | id ? ∈ dom user-info
| | info ! = user-info (id?)
| ---
|

----------SportsCenterClass----------
| | MaxPieceAllowed:\ N
| ---
| | borrowedby, previouslyborrowedby: PIECE ⇸ USER
| | EC: EquipmentClass
| | UC: UserClass
| ---
|
| -------INIT----------
| | EC.available ∩ EC.checkedout = ∅
| | UC.student ∩ UC.staff= ∅
| | dom borrowedby ⊆ EC.checkedout
| | ran borrowedby ⊆ UC.student ∪ UC.staff
| | dom previouslyborrowedby ⊆ UC.available ∪ UC.checkedout
| | ran previouslyborrowedby ⊆ UC.student ∪ UC.staff
| | ∀ user: USER | user ∈ UC.student ∪ UC.staff •
| | # borrowedby^{-1}(| {user} |) ≤ maxPieceAllowed
| ---
|
| -----CheckoutPiece----------
| | Δ (borrowedby, EC.available, EC.checkedout)
| | user ?: USER
| | piece ?: PIECE
| |----------
| | user ? ∈ UC.student ∪ UC.staff
| | piece? ∈ EC.available
| | # borrowedby^{-1}(| {user?} |) < maxPiecesAllowed
| | EC.available' = EC.available \{ piece ? }
| | EC.checkedout'=EC.checkedout ∪ { piece ?}
| | borrowedby' = borrowedby ∪ { piece? ↦ user? }

| | previouslyborrowedby' = previouslyborrowedby
| | UC.student' = UC.student
| | UC.staff' = UC.staff
| ---

| ----------EquipmentBorrowedBy----------
| | user ?: USER
| | out!: **F** PIECE
| | ----------
| | user ? ∈ UC.student
| | out! = borrowedby^{-1} (| { user ?} |)
| ---

| ----------PreviousBorrower----------
| | piece ?: PIECE
| | user !: USER
| |----------
| | piece ? ∈ EC.available ∪ EC.checkedout
| | {piece ? ↦ user! } ∈ previouslyborrowedby
| ---

| -----Return----------
| | Δ (EC, borrowedby, previouslyborrowedby)
| | piece ?: PIECE
| | user ?: USER
| |----------
| | piece ? ∈ EC.checkedout
| | user ? ∈ ran borrowedby
| | EC.available' = EC.available ∪ {piece ? }
| | borrowedby' = borrowedby \ { piece? ↦ user? }
| | EC.checkedout' = EC.checkedout \ {piece ? }
| | UC.student' = UC.student
| | UC.staff' = UC.staff
| ---

3

VITA

Yi Xie

Candidate for the Degree of

Master of Science

Thesis: USING FORMAL METHODS TO DESIGN AND IMPLEMENT AN OBJECT-ORIENTED UNIVERSITY SPORTS CENTER'S INFORMATION MANAGEMENT SYSTEM

Major Field: Computer Science

Biographical:

Personal Data: Born in Beijing, P. R. China on July 30, 1969, the daughter of Jinlai Xie and Zhaohua Xie.

Education: Graduated from Computer Institute, Beijing Polytechnic University in July, 1992; received Bachelor of Engineering degree in Computer Software Engineering. Completed the requirements of the Master of Science degree with a major in Computer Science at Oklahoma State University in May, 1997.

Professional Experience: Software Engineer, Hi-Tech Company, Computing Center, Chinese Academy of Sciences, 1992 to 1994; Graduate Teaching Assistant, Oklahoma State University, Department of Computer Science 1995.

Honor and Membership: Citgo Computer Science Scholarship; Member of The Honor Society of ***Phi Kappa Phi*** by election of the Chapter at Oklahoma State University.