1997

Object-oriented design and implementation of Ecosystem Information System (EIS) using Java

Petr Votava

The University of Montana

Object-oriented design and implementation of Ecosystem Information System (EIS) using Java

by

Petr Votava

B.Sc. University of Montana, Missoula, MT, USA, 1995

presented in partial fulfillment of the requirements

for the degree of

Master of Science

The University of Montana

September 1997

Approved by:

Chairperson

Dean, Graduate School

9-22-97

Date

10-15-97

Votava, Petr, M.S., September 1997 Computer Science

Object-oriented design and implementation of the Ecosystem Information System (EIS) using Java (72 pp.)

Director: Ray Ford, Ph.D.

The Ecosystem Information System (EIS) is used by ecosystem modelers to create, modify and access data repositories. The information in the EIS repository is organized hierarchically using object-oriented principles. Many problems associated with the current version of EIS led to an evaluation process that pointed out the need for the redesign of the current system in a more object-oriented fashion. The redesign of the EIS system was completed using Booch's [5] object-oriented methodology. The design was completed at a time when Java, a new object-oriented programming language from Sun Microsystems, was showing promising support for robust and platform independent implementation of object-oriented design. After risk assessment and testing, Java became the implementation language for a new version of the EIS system.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The Ecosystem Information System (EIS) is an object-oriented system that supports the creation of a distributed repository of ecosystem and natural resource information. The EIS repository is organized into hierarchies which are formed using an object-oriented framework, where classes contain data descriptions, instances are the datasets, and methods are the data transformations. One of the main goals of EIS is to provide data managers with a tool for data organization and dissemination which has both local and WWW interfaces, so that the database can be created and maintained locally, but certain parts can be shared with the outside world.

EIS has been built using object-oriented technology, specifically object-oriented analysis and design, and so the overall system is a collection of objects that interact with each other. This is in accordance with object-oriented analysis and design, which focuses on decomposing a complex system into a set of interacting objects. There are several problems that this type of design aims to solve. First it tries to reduce the complexity of large systems by focusing on objects, rather than on algorithms, as the building blocks for these systems. Second, it attempts to achieve flexibility of system implementation with respect to changes in requirements. This is accomplished through encapsulation and modularity. Last but not least, it tries to achieve a greater level of confidence in the correctness of the software, which in turn reduces the risks that are inherent in developing complex software systems.

But object-oriented design and analysis cannot be viewed separately from the evolution of the whole software system. This overall process is best described by a model such as Boehm's spiral model [6] which shows different stages of the development as being revisited in an iterative fashion. This is in contrast to the traditional waterfall model, where each stage of development is expected to be visited only once. Thus, the spiral model more closely reflects the reality in which requirements can change many times during the development process, and consequently some parts of the design and implementation must also change several times during the lifetime of the system.

1.2 Problems

Although the current implementation of the EIS is already being used, it is still in a nonterminal stage of the development process according to the spiral model. There are several problems with the current system (labeled "version 2.2") that limit its usability in many ways.

First is the problem of *maintainability*. There are almost 70,000 lines of code in the system, with almost no documentation other than high level design descriptions, so it is getting to the point where it is very hard for one or more developers to maintain the whole system. Second is the problem that the object-oriented design was translated into a more traditionally structured implementation. Mixing these two methodologies on such a large system has caused numerous problems during the system maintenance, particularly when making modifications due to changes in requirements. These deficiencies are not so much the fault of the designers and developers of the original system, as much as problems with the tools that were available to them. The worst problems were caused by the graphical user interface (GUI). This was entirely written in XMotif which does not interface well with C++ classes. Moreover, the GUI code was generated using the Teleuse GUI generator, which generates C code in a somewhat cryptic structured manner. The result is that the GUI is very difficult to modify and maintain manually, due to its large size (almost 45,000 lines) and unusual structure. Lex and yacc also do not have particularly good interfaces to C++ and since they were used to implement the EIS language processor, the code from the language processor also fails to match the object oriented design structure. Last, Sun RPC was used to implement a communication layer between server and clients, and was the source of many problems related to robustness and security explained below.

Another problem embedded in the current version of the system is a set of major *security* holes. These would enable a potential attacker to perform read/write operations on any files that have the same ID (user) or GID (group) as the user running the server. In order to assess the severity of this problem, I experimented with simple "attacks", and was able to easily download the password file and overwrite the .rhosts file on a system running the EIS server. These security problems definitely deserve attention, because of the EIS requirement that the system be distributed over the network.

Another of the big issues with all large complex software systems, including EIS, is the problem of *portability*. In the current version of EIS only portability with other Unix systems was addressed during the development. As a result, EIS would be extremely hard to port to run on either a PC (under Windows95 or WindowsNT), or on a Macintosh computer.

Other aspects of a software system that are very important from the user's point of view are *robustness* and *error recovery.* In other words, how often does the program crash and what are the results of such a crash? Clearly, users will be very reluctant to use a system which wipes out its entire database when it crashes. The current version of the EIS system has some robustness problems resulting from RPC problems, and also because it was implemented in C/C++ without explicit support for exception handling.

The final problem of concern arises with the allocation of resources in the current version of the system. The main problem is that large blocks of memory are allocated on the startup of the system, regardless of their usage. This poses a severe limit on the number of objects that the system can use, which can become a problem when the system is running on a machine with limited resources (especially a small amount of memory).

1.3 Proposed Solutions

First, it is important to realize that we are entering another iteration of the spiral model in the development of EIS. The maintainability of the system can be improved dramatically by using the object-oriented design to redesign components which have been identified as problem areas in the evaluation of the current system. Note, that the entire system does not have to be redesigned, but there are several parts that are specifically problematic in the current system. These include the GUI, symbol table, language processor, and the entire WWW interface.

Several general problems with EIS also need to be addressed. There are several possibilities for dealing with the security problems. The first one is to maintain the system in a single-user mode. In that way no part is directly accessible through a network, and so the system cannot be compromised (this is essentially how EIS is currently being used). Given this approach, it is still possible to globally distribute EIS information by extending the WWW interface to register hierarchies that were created locally with a designated EIS/WWW server, which then takes care of the distribution using a standard HTTP protocol. This seems the most feasible way to fix security problems. Other, more sophisticated alternatives exist, mainly implementing some kind of encryption/digital signature system in both the EIS client, and EIS server, using a scheme such as JavaBeans or Netscape SSL. It is my opinion however, that these alternatives would be an "overkill" based on what we are trying to achieve.

The problem of portability can best be solved by changing the language for the system implementation to a language and corresponding tools and libraries that are more platform independent than C/C++ with XMotif. The choice of today's developers seems to be Java, which uses platform independent libraries for key functions (including GUI management) and creates platform independent bytecode. Even though the language is still in development, it exhibits numerous features that seem to fit very well into the overall requirements on the system. More details on the recommendation to use Java as the language for the next version of EIS implementation are given below.

The proposed solution to resource allocation problems is to redesign the system with emphasis on dynamic rather than static allocation of resources, and with more attention facing towards garbage collection and general resource management.

1.4 Why Java?

There are many reasons why I recommend use of Java for the implementation of the new version of EIS. First is a strong connection between Java, object-oriented design and object-oriented programming. Java supports all aspects of object-oriented design and clearly exhibits all the properties of object-oriented programming language, which makes it a strong candidate for the language of choice. It also addresses the key portability issues already mentioned, which seem very important in today's world where the more systems the application can run on, the better. Next is the feature of foremost importance, Java's ease of integration with World Wide Web facilities. In moving EIS from a distributed system to a system which attains its distribution facility through links to the WWW, the integration between EIS and the Web becomes critical. Since Java programs can be packaged as either applications (running locally) or applets (running remotely through Web browser), a lot of work can be saved in designing the Web interface from a scratch to replace the current prototype of "cgi bin" scripts. Last, but not least, Java is simple relative to C++, which should result in simpler code, easier maintainability, and ability for the implementer to focus less on the implementation details and more on consistency with the design. Also since Java has become extremely popular over the last several months, a large number of tools and libraries exist for this language which exhibit features of portability, simplicity, and so on.

Chapter 2

Background

2.1 Object Oriented Design Definition

In every engineering discipline the purpose of a design is to provide an intermediate generic representation of requirements that will map easily into implementation. In this sense, software engineering is no different. However, in the case of software engineering the design cannot be viewed separately from the rest of the development process, because of issues that are specific to software development. One of the main differences that makes the software development process unique is the fact that the requirements can change over the lifetime of the system, and the design (and the designer) has to be able to deal with it. Other related difficulties result from the rapidly changing base technology, the perception that software is easy to change, and inherent complexity of software systems.

Over the years there have been numerous design methods that have gained popularity, ranging from top-down structured design to data-driven design. Aspects of all of these design techniques have been combined and evolved to form the object-oriented design method. We can start with the formal definition, which states

"*Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design*" [5]

The most important part of the definition is the notion of object-oriented decomposition, which makes object-oriented design very different from other approaches. Classes and objects, rather than algorithms or procedures, are used as the basic building blocks of the design. To restate the definition in other words, object-oriented design focuses on identifying classes and objects and their relationships in a complex system.

But what is an object? What is a class? Every object can be identified by three attributes: *state*, *behavior*, and *identity*. The *state* of an object is determined by its static properties (for example the types of its variables) and dynamic, current values of each of these properties (for example, values of its variables). It is good practice to encapsulate the state of the object and provide methods for its manipulation, rather then providing direct access to its properties. This has the advantage that it makes the application design independent of the object's internal representation, which makes the design more flexible. The *behavior* of an object represents its visible and testable activity. In other word it defines how an object is perceived to act and react in terms of its state change, when actions are performed on it from the outside. Lastly, the *identity* of an object is a property that distinguishes the object from any other object.

The concept of object and class is tightly related, because each object is an instance of some class. However, there are also important differences between an object and a class. An object is a concrete entity that exists in time and space, but a class is an abstract description of the characteristics common to its objects. So in other words," ... *class is a set of objects that share a common structure and a common behavior*.[5]" Another difference is that classes are mainly static, that is their existence, relationship, and semantics are fixed before execution of a program, but objects are created and destroyed dynamically during the program execution. From the designer's point of view a class is an important entity, but class descriptions alone do not constitute the design of the system. Since the classes do not exist in isolation, it is necessary to identify the relationships among the classes and the objects in the system. This is very important step in object oriented design, because these interactions define the overall behavior of the system.

Though different object-oriented design methodologies differ on the details, most agree that there are six different kinds of relationships that can be used to describe critical class/class, class/object, and object/object interactions: *association*, *inheritance*, *aggregation*, *using*, *instantiation*, and *metaclass*. The discussion below uses the specific interpretation given by Booch[5] for each of these relationships. Note that each relationship has a formal set-theoretic definition, but we generally give a more intuitive description here.

Association refers to a bidirectional semantic dependency between two classes. For example, there is an association between students and courses offered by school, which in terms of cardinality is many-to-many association, meaning that each student is taking many courses and each course is attended by many students. *Inheritance* is a relationship among classes, in which a subclass is identified as one which shares all the structure and behavior of its superclass (or parent class), though the subclass may have additional properties not possessed by the parent. *Aggregation* defines the whole/part relationship between classes, where instance of one class (C_1) can be an attribute of another class (C_2). In this case class C_1 is the *part* of the class C_2 which represents the *whole*. A client-server interaction is depicted in the *using* relationship, in which one class is requesting services of another class. Each of the above is a class/class relationship, which means that if there is a relationship **R** between classes C_1 and C_2, then any object O_1 of C_1 and O_2 of C_2 have the same relationship **R**.

To add a higher level of abstraction and genericity, *instantiation* and *metaclass* relationships are used. *Instantiation* is a relationship between a parametrized class (also called a generic class) and its instances. A parametrized class is an abstract class that must be instantiated before objects can be created, and so it serves as a template for other classes. Last of the relationships mentioned above is *metaclass*, which is a class whose instances are themselves classes. This relationship treats classes as objects that can be manipulated.

In object-oriented design, the goal is to identify classes and objects, and their relationships, from the vocabulary of the problem domain. Most of the time these key concepts are represented by nouns in any descriptive text. This differs from structured algorithmic design, in which we first look for the active verbs in the description of the problem domain, which identify the flow of execution. When considering the object-oriented design of a complex system, there are two main tasks to be completed by the designer. First, the designer must identify the classes and objects from the vocabulary of the problem domain. Second, the designer must identify the relationships among the classes and objects that express the requirements of the problem. In the terms of implementation, the classes and objects are called the *key abstractions*, and the relationship structures are the *mechanisms* of the design and implementation.

So what are the overall reasons for using object-oriented development over classical structured development? The main benefits come from the characteristics of the object model, which exploits the expressive power of object-oriented programming languages, encourages the reuse of software components, leads to systems that are more resilient to change, reduces development risk, and appeals to the working of human cognition [5].

However there are also two main drawbacks in using object-oriented methodology. First, the performance cost related to the communication overhead between two objects in object-oriented programming language can be higher than a function invocation in a procedural language. Second is the inherent cost of switching to a new technology. Despite of these drawbacks, the benefits of object-oriented technology usually far outweigh the risks associated with the drawbacks mentioned above [5].

2.2 A Spiral Model of Software Development

Object-oriented design and analysis cannot be, however, viewed separately from the evolution of the whole software system. This overall process is best described by a model such as Boehm's spiral model [6].

The spiral model is an example of an software process model, which describes the order of the stages to be completed during the process, the transition criteria for advancing from one stage to another, and the repetitive or iterative nature of the process/subprocesses. Software process models are especially important for large development projects, because they function both as management and descriptive tools that specify the order in which the major tasks should be performed, and that allow information about what was done to be placed in proper context. Many different models of software development have been proposed, and they have all gradually evolved towards a form like that used in the spiral model or its derivations.

What makes spiral model so different from traditional process flow models? The main difference is that the spiral model is risk-driven rather than document- or code-driven. It also eliminates many problems associated with the other models while taking advantage of their strengths. A typical cycle of each spiral consists of several activities. First, the designer identifies an objective of the next portion of the product, such as performance or robustness. Next, he evaluates the alternatives for development of this part. These alternatives can be for example, design, reuse, or purchase the part in question. Next, the designer identifies and evaluates the constraints associated with the alternatives. The next step is to evaluate the alternatives with respect to the objectives and to the constraints, identify areas of project risks, and evolve a strategy for resolving these risks. Finally, the designer chooses an approach and executes and evaluates appropriate task.

This risk-driven approach to each subset of the spiral model steps allows the designer to choose the particular software development approach that is best suited for this project, or even for just this phase of development whether it is specification-oriented, simulation-oriented, or prototype-oriented. This implies that most of other software development models can be accommodated within the process flow of the spiral model.

Finally, the way that the spiral model deals with maintenance phase of the software development process differs substantially from the approach of most other models. In many models the maintenance phase is separate from the rest of the development flow, which lumps a potentially vast set of activities involving changing specifications, redesign, and re-implementation into "maintenance". In the spiral model, maintenance is simply an ongoing spiral (or spirals) in which the specifications, design, and implementation issues are continually reevaluated in the changing context. In practice, using the spiral model requires the designer to overcome several difficulties, such as matching the spiral to details of contract software, developing risk-assessment expertise, and the further elaborating spiral model steps [6]. Despite the problems described above, I think that the benefits of the spiral model outweigh the difficulties, because of the risk-driven nature of the model that enables us to evaluate our options before we start each phase, and thus preventing us from costly changes into the design later.

2.3 Ecosystem Information System (EIS)

The Ecosystem Information System (EIS) is an object-oriented system designed to support the creation of repositories of ecosystem and natural system information. EIS allows data managers to build an index to a heterogeneous collection of datasets in an intuitive (object-oriented) fashion. It also provides assistance in translating this index into a Web accessible form. For the data manager, EIS indices formalize relationships between datasets using traditional hierarchical classification principles long used in biological and spatial systems. For the user, EIS hierarchical indices provide a structure that supports use of standard Web software to browse and query the collection of datasets.

EIS indices are constructed using traditional hierarchical classification principles, expressed in terminology taken from object-oriented modeling techniques. A *class definition* identifies the properties that are unique to a particular type of dataset. A *class hierarchy* defines an *inheritance relationship* among classes. Class descriptions thus hierarchically organize the shared and unique properties of various datasets. Once this classification framework is established, a specific dataset can be attached as an *instance* of a particular class. A dataset transformation is defined generically as a function on class instances. An implementation of this logical function is called a *method.*

A modeler builds an index using the following approach. First, he identifies classes of datasets that share well-defined properties and lists these properties. Second, the modeler identifies relationships among classes based on which properties are held in common and which are unique. Next, he registers each dataset as a member of a particular data class, and finally, he registers each program that manipulates dataset as a method of a function defined on a particular class of datasets.

2.4 Spiral Model in EIS Development

The development process for EIS has been managed and can be described using the spiral model. The first design started in 1993. Since then several spirals have been completed. A diagram of the entire process is shown in Figure 2-1. The first spiral was completed when the first prototype was finished. This system featured a simple grammar for class and object descriptions, rudimentary language processing, with no static semantic restrictions, no user interface, no distributed objects, and no Web integration [2].

In the second spiral of development, this prototype was improved by adding a user-friendly graphical interface [3], and by looking at how the system could be extended to accommodate distributed objects. This scheme was designed, partially implemented and integrated in the second prototype [3]. A weakness of the second prototype was that the processing of the EIS description language remained partially incomplete, lacking effective static semantic constraints imposed on the hierarchies created by the system.

In the third spiral, the language deficiency was "fixed" formally by definition of an attribute grammar for the system, which extended the language to include the formal definition of semantic constraints. The third prototype integrated this checking with GUI enhancements and other improvements [1]. This version was the first real release of an executable system that was used outside of the department. However, various security problems were embedded in the implementation of the distributed part of the system, and it was not suitable for general release.

The project described here starts at the beginning of the fourth spiral. The main requirement at the beginning of this stage was to extend the EIS functionality to provide an interface to the World Wide Web. A partial version of a Web interface was created and embedded in the third EIS prototype, using C, C++ and CGI-scripts. However, with the emergence of Java as a platform independent, Web interface programming language, it seemed worthwhile to try to evaluate its importance in EIS evolution. Thus, this project

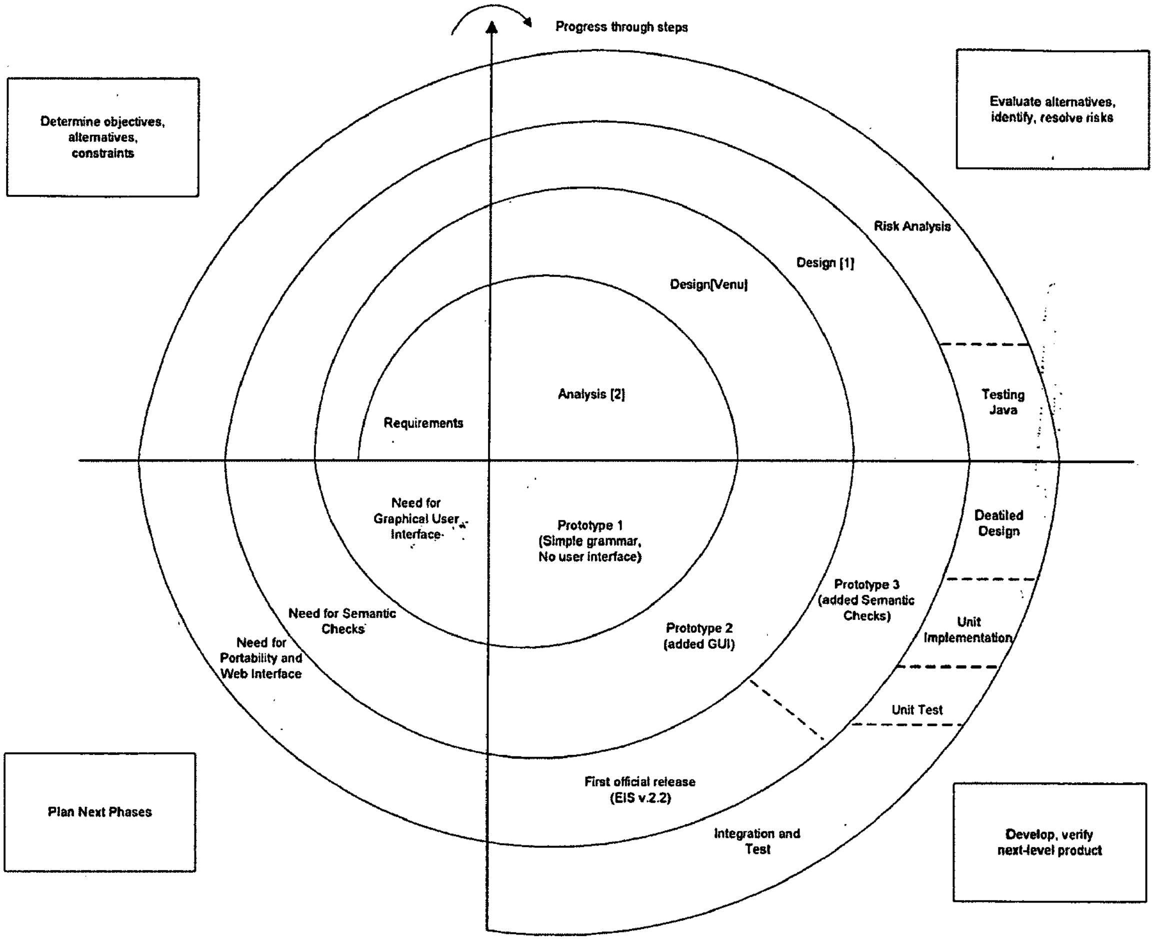
Progress through steps
Determine objectives, alternatives, constraints
Evaluate alternatives, identify, resolve risks
Risk Analysis
Design [1]
Design[Venu]
Analysis [2]
Requirements
Testing Java
Need for Graphical User Interface
Prototype 1 (Simple grammar, No user interface)
Deatiled Design
Prototype 3 (added Semantic Checks)
Unit Implementation
Need for Semantic Checks
Need for Portability and Web Interface
Prototype 2 (added GUI)
Unit Test
Plan Next Phases
First official release (EIS v.2.2)
Integration and Test
Develop, verify next-level product

Figure 2-1
Spiral Model of the EIS Development

starts with an evaluation of the feasibility of changing the implementation of EIS to Java. It seemed that with the features of Java, which will be discussed in more detail in Chapter 4, and the variety of tools that are available for development of Java-based applications, the re-design and re-implementation of the EIS system was feasible, and would provide significant long term advantages to ongoing EIS development.

Chapter 3

Object-Oriented Design of Ecosystem Information System (EIS)

3.1 Notation

3.1.1 Views of the System

Before starting to describe the design of the Ecosystem Information System (EIS), it is necessary to introduce the notation that will be used throughout this chapter. Notation is a very important part of any design, whether it is in the form of blueprints in civil engineering, or in the form of diagrams in software development process. Its main purpose is to present the design in a manner that is more formal than written description, easier understood than source code listing, and more generic than a programming language.

Each software system can be described in several different ways. First, there is the *logical* view, which describes the abstract composition of the system in a form of key abstractions and mechanisms that logically define the system. Second, there is the *physical* view, that describe the concrete hardware and software composition of the system. There is, however, another dimension in describing any software system. Since software systems are dynamic systems, it is appropriate to distinguish between describing the system's structure (*static* view) and the system's behavior (*dynamic* view). These two views are complementary, because the behavior of a system cannot be defined without structure, and likewise structure by itself does not tell us much about the dynamic behavior.

This chapter concentrates on describing the logical view of the system in both static and dynamic forms using the form of Booch[5] diagrams whose notation is briefly described below. The current implementation of the system is described in greater details in Chapter 4.

3.1.2 Booch's Notation for Object-Oriented Design

As described above, there are different views of the system, and thus there are several different types of diagrams. The presentation here uses four of the diagrams described by Booch, namely:

- Class Diagrams
- Object Diagrams
- State Transition Diagrams
- Interaction Diagrams

3.1.2.1 Class Diagrams Notation

Class diagrams are used to show the existence of classes and their static and logical relationships within the system. The class icon is shown in Figure 3-1, in a form of "cloud" with dotted borders to indicate the abstractness of a class definition. Each class must have a name, and may have a set of attributes, operations, and constraints. An attribute may have a name, a class, or both, and optionally a default value. The notation for these entities is as follows:

• A	Attribute name only
• : C	Attribute class only
• A : C	Attribute name and class
• A : C = E	Attribute name, class, and default expression

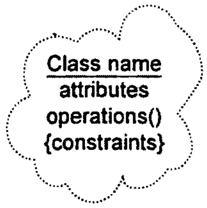

Figure 3-1
Class Icon

Operations are presented in following manner:

N()	Operation name only
R N(Arguments)	Operation name, return class, and formal arguments

Another type of class is an abstract class, whose icon is depicted in Figure 3-2. Since classes are hardly ever isolated in a system, we need some means to show the relationships among classes that were discussed in Chapter Two. These relationships are shown in Figure 3-3. In addition, the linkages for association and aggregation can be adorned with the cardinality of the relationship as follows:

1	Exactly One
N	Unlimited number
0 .. N	Zero or more
1 .. N	One or more
0 .. 1	Zero or one
3 .. 7	Specified range
1 .. 3, 7	Specified range or exact number

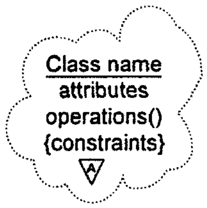

Figure 3-2
Abstract Class

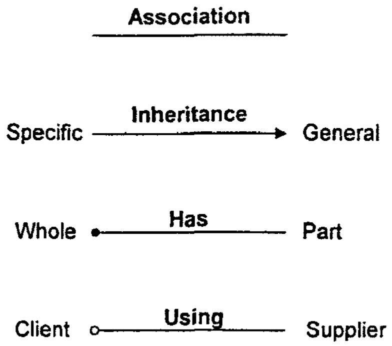

Figure 3-3
Class Relationship

In a large systems there are usually so many classes that the class diagram can become overwhelming. To reduce details, the system view can be presented in the form of higher level class diagrams using "class category" icon shown in Figure 3-4. A class category is essentially a cluster of classes logically grouped together. There is a direct correspondence between the class category in the Booch's notation on the abstract level and the concept of packages (of classes) in the Java programming language on the physical level. The relationships among the class categories are depicted in the form of "using" notation, denoting one category's dependencies on the other class' categories.

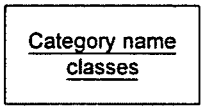

Figure 3-4
Class Category

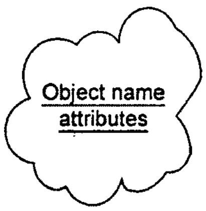

Figure 3-5
Object Icon

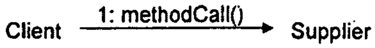

Figure 3-6
Object Relationships

3.1.2.2 Object Diagrams Notation

Object diagrams are used to show the existence of objects and their relationships in the logical design of the system. In other words, object diagrams are used to represent a set of possible interactions among objects in the system. The icon for object is shown in Figure 3-5. The names of objects follow the same convention as names for attributes, namely :

A	Object name only
:C	Object class only
A : C	Object name and class

The notation for object relationships is shown in Figure 3-6. Because objects communicate via method invocations, the relationship shows the direction of the invocation using an arrow, labeled with the operation invoked. Generally object diagrams show entire set of possible invocations among the objects. Optionally, sequence numbers can be shown on the arc in diagrams that try to show the sequence of events in a particular scenario.

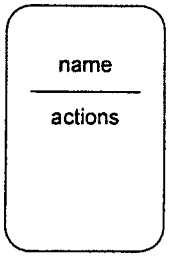

Figure 3-7
State icon

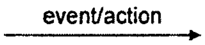

Figure 3-8
State Transitions

3.1.2.3 State Transition Diagrams Notation

State transition diagrams are used to describe dynamic behavior of individual objects. They show the external events that cause an object to change its internal state, and the internal actions that result from this change. There are two main parts to these diagrams: states and state transitions. A *state* (Figure 3-7) represents the cumulative results of object's behavior. Each state should have a name that is unique for all the states within this particular object, and an optional list of actions associated with this state. A *state transition* (Figure 3-8) represents a change of state in an object. The change is usually triggered by some event, and subsequently an action is performed which changes the internal state of the object.

3.1.2.4 Interaction Diagrams Notation

Interaction diagrams are used to trace a particular sequence of object interactions that occur during execution of particular scenario. The main difference between an object diagram and an interaction diagram is that in an interaction diagram it is easier to see the exact order of messages exchanged between objects. The reason for using both diagrams is that interaction diagrams do not scale well when the number of objects increases -- object diagrams are much more readable in this case, because they omit the ordering details.

3.2 Design

3.2.1 Identification of Classes and Relationships

As described in Chapter Two, the key to object-oriented design is identification of relevant classes and their relationships. This task has been first started during the analysis phase of this project, and it has been described by Ron Righter in [2].

The EIS system has been described in the previous chapter. This description was, however, done from the user's point of view. This is useful for the analysis, but is not sufficient for more detailed description of the internal structure of the system. This section describes the classes and objects that were identified during the analysis and their function in the system. The complete formal description of the crucial classes is covered in Appendix A.

From the system requirements we know that all EIS operations are performed on objects that are part of a hierarchy, so *hierarchy* is an ideal candidate for a class. In order to assure correct syntax of hierarchies, a context free language was developed to specify the EIS syntax. The grammar was then used to automate language processing in the form of a parser which accepts a proposed object description and verifies its syntax. Thus *parser* is another important object in the EIS system. It is, however, just a tool for enforcing the correct syntax. In order to store the structure of the objects of the hierarchy in a persistent way, *syntax tree* objects are generated and saved by the parser. In addition to syntax structure of the objects in the system, a set of semantic rules has been developed to provide for better system integrity [1]. The semantic rules, which are concerned with consistent use of identifiers defined in the hierarchy, can be easily checked by consulting *symbol table* objects that are also generated by the parser. But what is the input to the parser? The input is an object that represents a textual *description* of the object created by user. Lastly, a user has to be able to create these object descriptions in some way. To aid the user in doing so, a *graphical user interface (GUI)* is provided, which leads the user through the process of creating the objects and organizing them, as described in Chapter Two.

3.2.2 Class Diagrams

In order to better depict the higher level design of the system, the top-level class diagram in Figure 3-9 shows the relationship among the class categories. The relationships shown in this figure are package "using" dependencies rather than specific class relationships. The main driver of the system is the graphical user interface (*GUI*), which is used to fill *descriptions* of objects, pass these to the *parser*, which generates *symbol table* and *syntax tree* objects that are subsequently stored in the hierarchy. In case of a syntax or semantic error, the parser or hierarchy generates an *exception* which is displayed to the user through the GUI.

Figure 3-10 shows class diagram for the EIS Hierarchy in more detail. The *Hierarchy* class consists of a set of instances of *HierarchyNode*, each of which has exactly one instance of *SyntaxTree* and exactly one instance of *SymbolTable*. The hierarchy class is further associated with the *SemanticError* class, because it can throw a semantic error exception when performing semantic checks. It is also associated with *HierarchyIO* class, which provides the interface between the hierarchy and the filesystem.

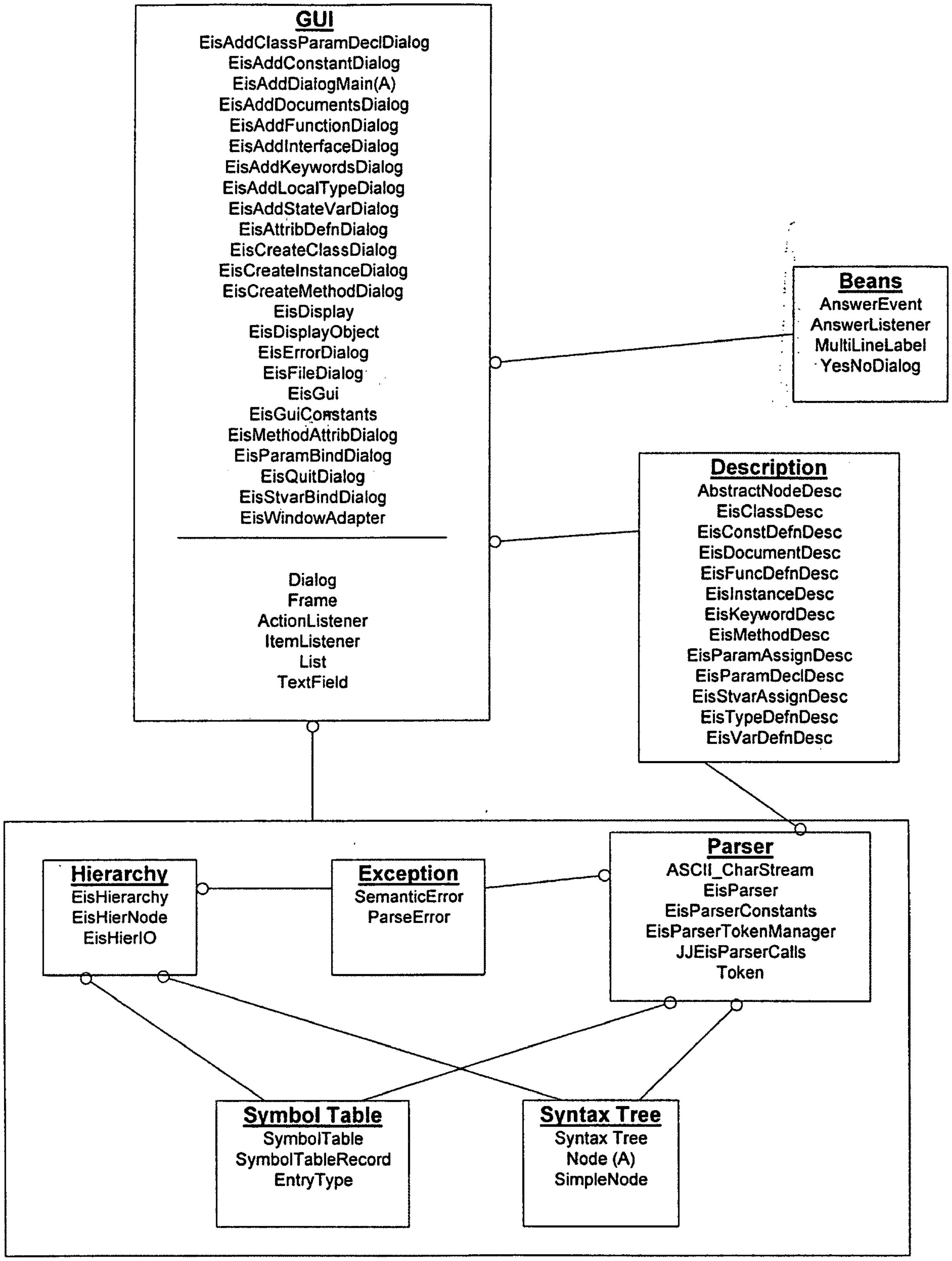

Figure 3-9
Ecosystem Information System Top-Level Class Diagram

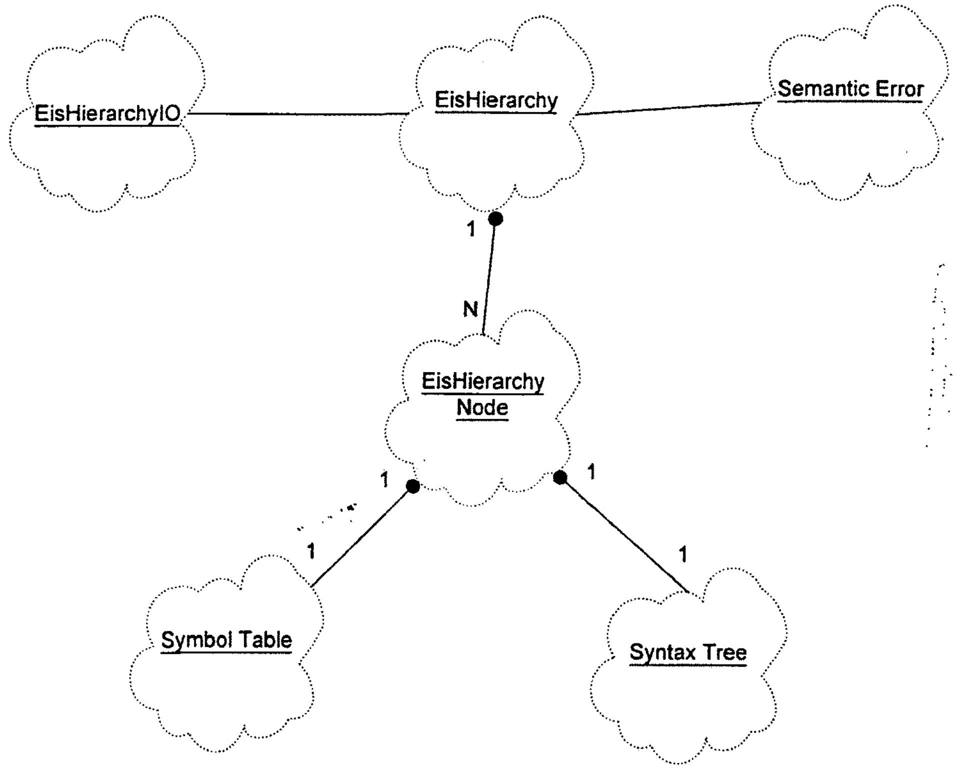

Figure 3-10
EIS Hierarchy Class Diagram

A more detailed view of the *SymbolTable* and *SyntaxTree* classes is depicted in Figures 3-11 and 3-12, respectively. The *SymbolTable* class is an aggregate of instances of *SymbolTableRecord*, and is associated with the *EntryType* class that defines constants used by the symbol table. Similarly, the *SyntaxTree* class includes multiple instances of *SimpleNode*, which is a subclass of an abstract class *Node*. Both *SyntaxTree* and *SymbolTable* classes are used by the *Parser* class.

Because the *GUI* class serves mainly as a driver for the rest of the system, it is not of great interest from the design point of view, and so the last class category described in the form of class diagrams is the *Description*. The three main classes in this category are *EisClassDesc*, *EisInstanceDesc*, and *EisMethodDesc*. Instances of these classes serve as a

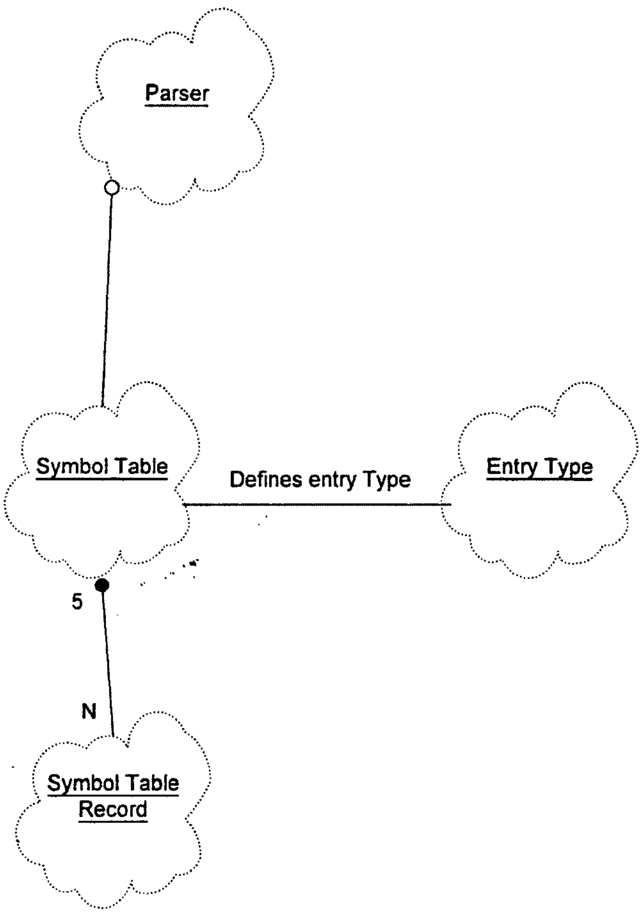

Figure 3-11
Symbol Table Class Diagram

textual representation of key objects used in the system, EIS classes, instances, and methods. Since all three classes share several attributes, namely keyword and document descriptions, they are subclasses of the *AbstractNodeDesc* class that aggregates these common attributes. Additionally, *EisClassDesc* aggregates classes that serve as descriptions of constants, functions, variable definitions and assignments, parameter definitions and assignments, and type definitions. Similarly, instances of *EisInstanceDesc* include descriptions of variable and parameter assignments. The structure of this class category is shown in Figure 3-13. Class diagrams in Figures 3-9 through 3-13 represent a static logical view of the system structure without any reference to its behavior. The dynamic view of the system is described in subsequent sections and diagrams.

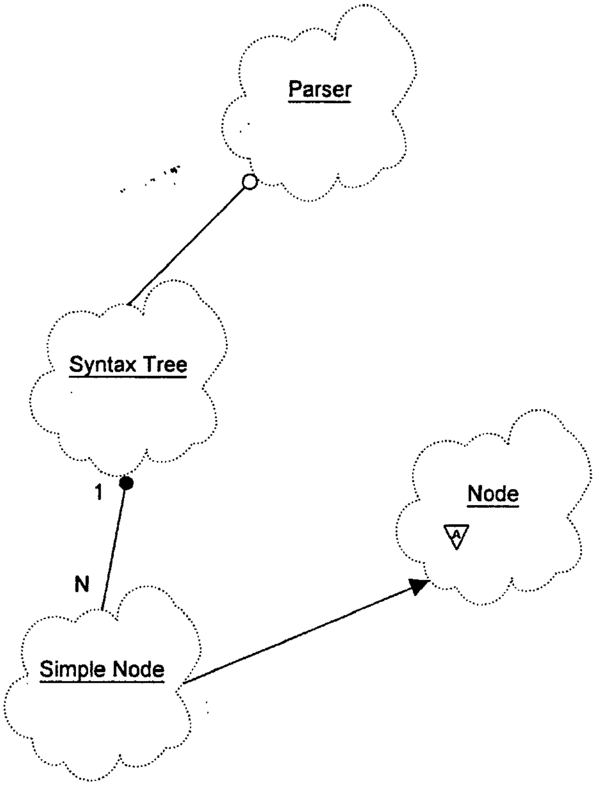

Figure 3-12
Syntax Tree Class Diagram

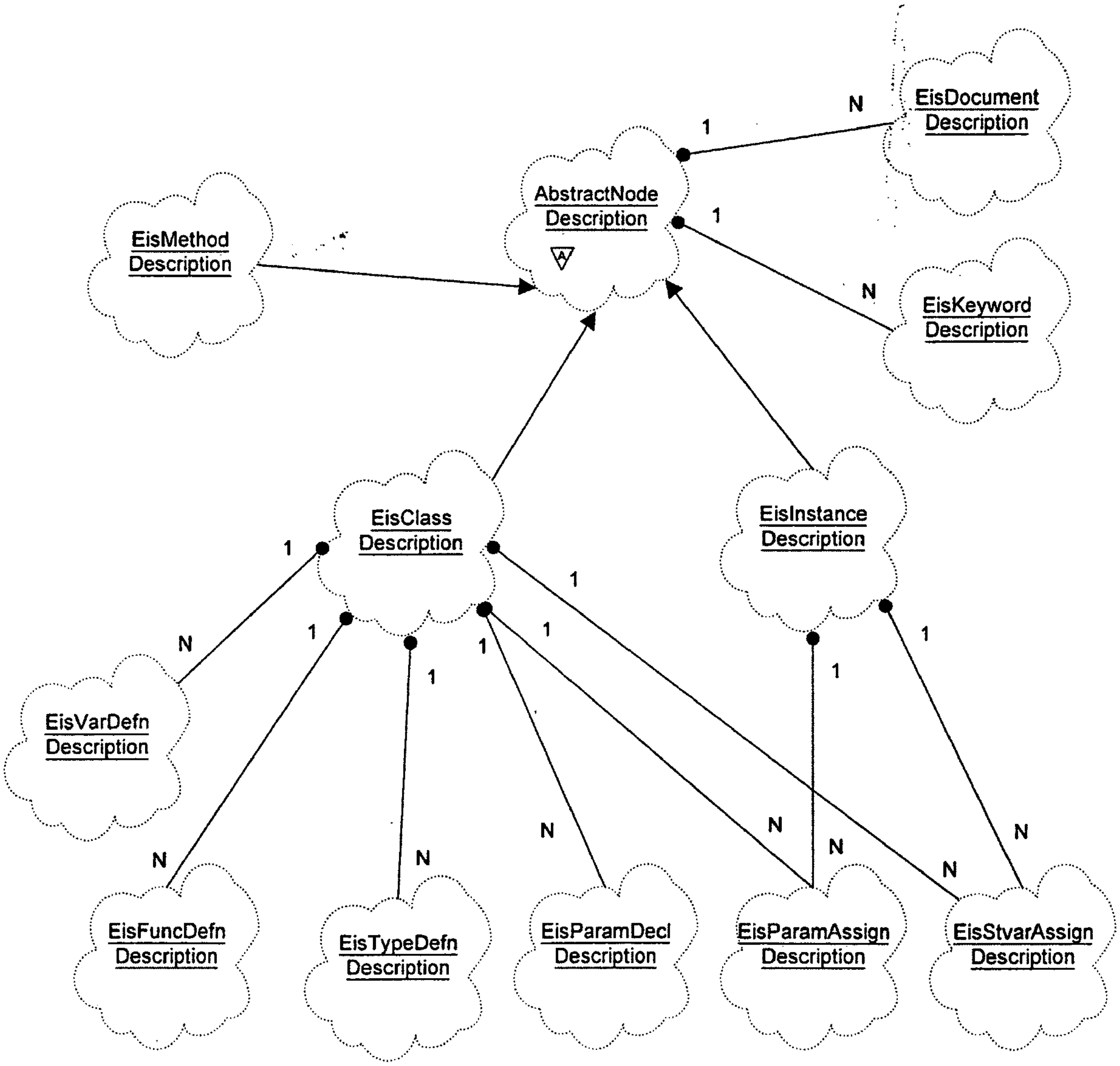

Figure 3-13
Class Diagram of Descriptions of EIS Objects

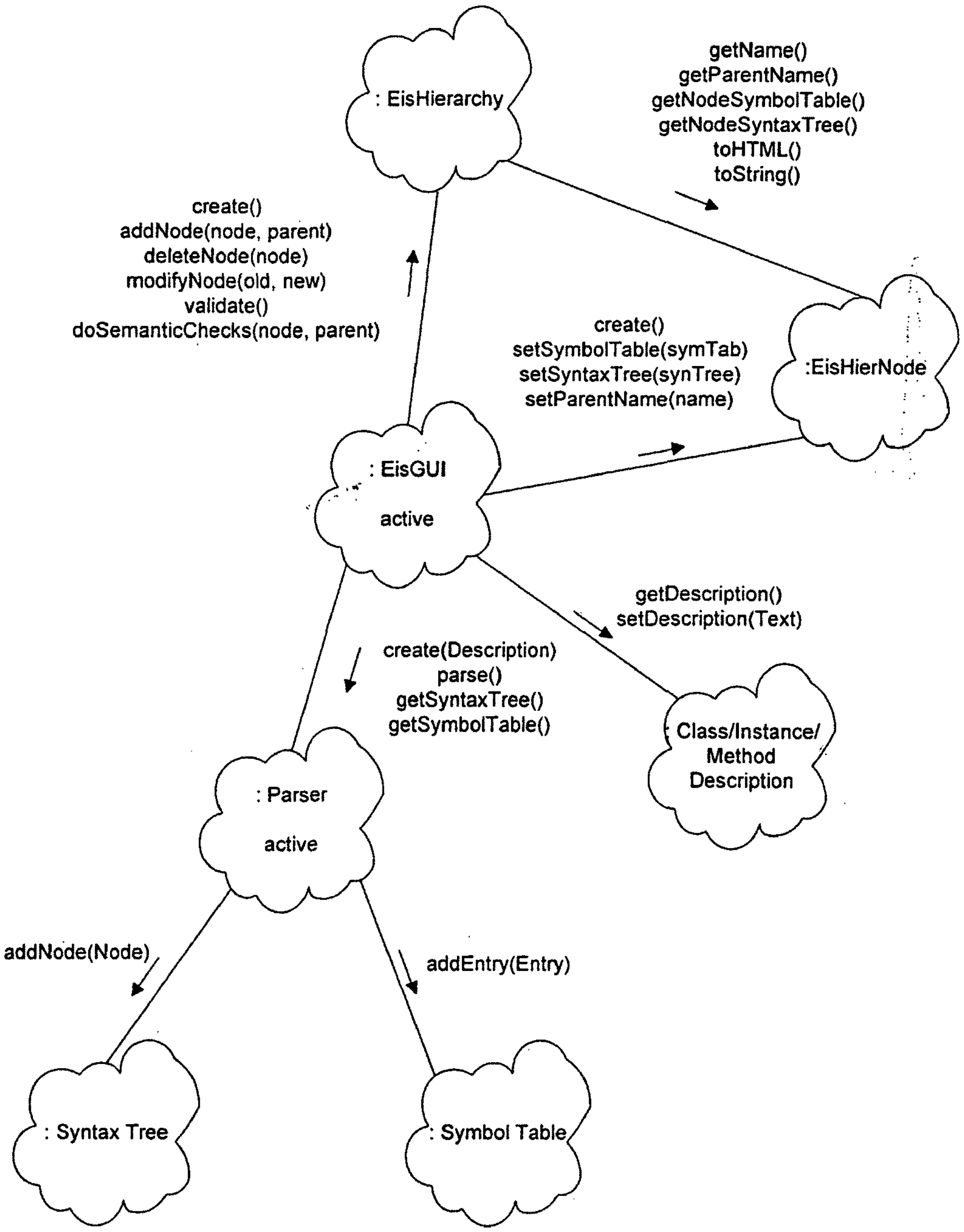

Figure 3-14
Top-Level EIS Object Diagram

3.2.3 Object Diagrams

Object diagrams shows the existence and the relationships among the class instances in the system. These diagrams can be divided into two classes. First, there is the top-level object diagram, which represents a high level view of the system. This object diagram is shown in Figure 3-14, and it shows *all* the possible messages that are exchanged among the objects in the form of method invocation.

From Figure 3-14 we can see that there are two active objects in the system, *EisGui*, and *Parser*. *EisGui* is the user interface object that leads the user through the process of manipulating the EIS database. The user can fill out the object *Description* through the *EisGui* object, and *EisGui* subsequently transfers the *Description* together with the control to the *Parser* object. *Parser* checks the syntax of the *Description* object, and generates the *SyntaxTree* and the *SymbolTable* objects by adding nodes and symbol table entries to them. When the *Parser* is done, control is transferred back to the *EisGui* object, which can query the *Parser* and extract the *SyntaxTree* and the *SymbolTable* objects and perform a user specified operation on the *EisHierarchy* object.

A second type of object diagrams shows only that subset of objects and messages that are specific to a certain scenario, for example showing the flow of control and data among objects that participate when an object is added to a hierarchy. As mentioned at the beginning of this chapter, the information depicted in these diagrams is almost identical to the information shown in the interaction diagrams, which are used later in the chapter to present all important scenarios.

3.2.4 State Transition Diagrams

State transition diagrams show the state space of a given class. This section presents the state transition diagrams for the most important classes of EIS. They are:

- EIS Hierarchy
- Syntax Tree
- Symbol Table
- Parser
- Graphical User Interface (GUI)
- Description
- EIS Hierarchy Node

3.2.4.1 EIS Hierarchy

The state transition diagram of the EisHierarchy is shown in Figure 3-15. This is probably the most complex state transition diagram, because it shows all the states of the most important class in the system. At system start, the hierarchy is *empty*. It can become *initialized* (or *non-empty*) by either going through the *opening* stage, or by creating a new hierarchy and filling in initial information. From the *initialized* state, it can *add*, *modify*, *delete*, *import*, *save*, or *export* nodes. The meaning of these states are fairly obvious. When a node is added or modified, it is necessary to *perform semantic checks*. If semantic checks do not fail, control is transferred back to the previous state. However if there is a semantic error, a *failure* occurs and the user is notified about the problem. Note that the import state can not follow directly to the semantic checks stage, because first all nodes of the hierarchy have to be imported, and then during *validating* all the nodes must be checked for semantics.

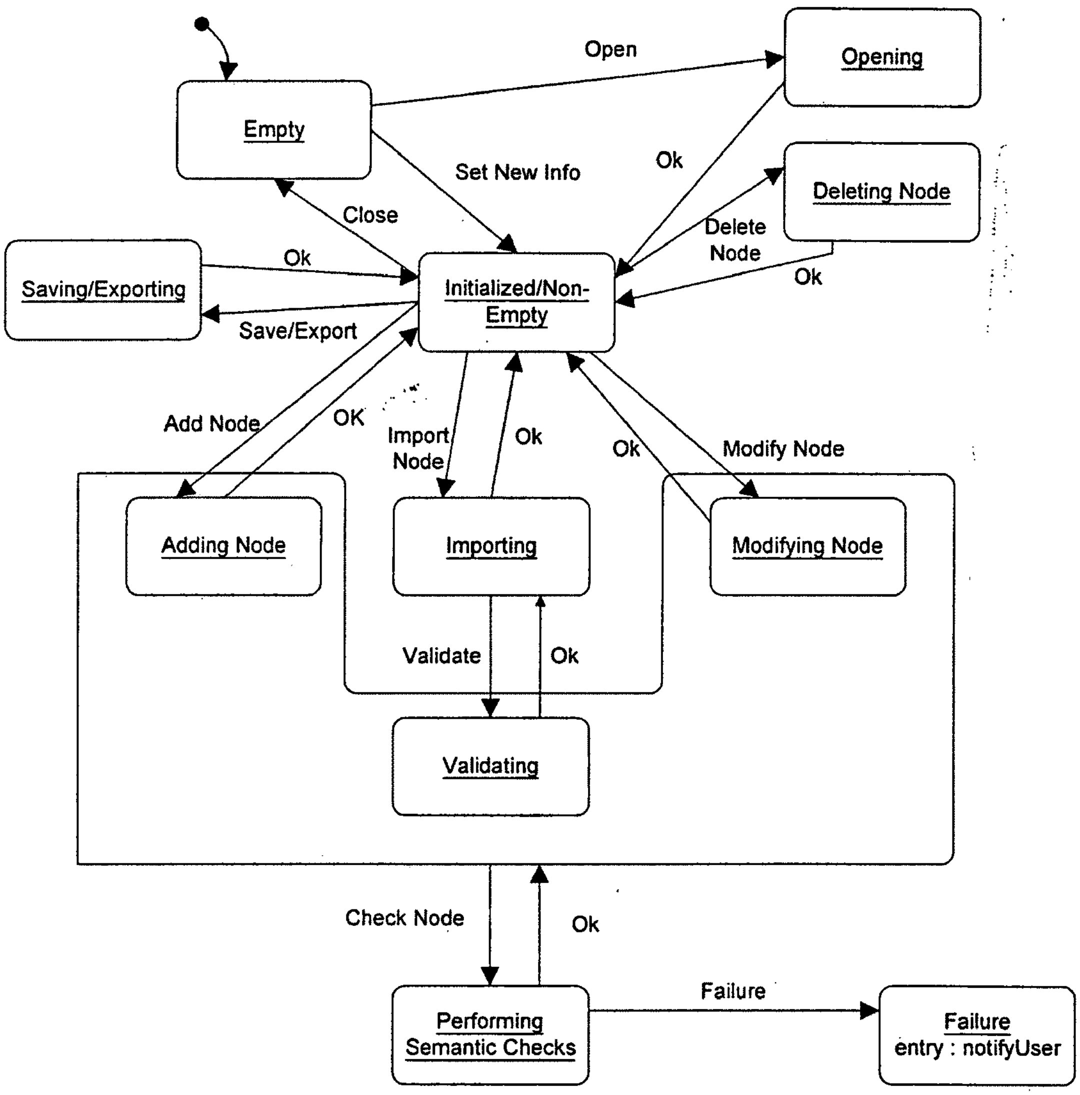

Figure 3-15
State Transition Diagram for EIS Hierarchy

3.2.4.2 Syntax Tree

The state transition diagram in Figure 3-16 shows the states and state transitions of the *SyntaxTree* object. When created, the syntax tree is *empty*. It becomes *non-empty* through the process of *adding* nodes. If a *failure* occurs during the addition of a node to the tree, the error is reported to the user. An important state of the tree is when it is traversing its nodes, *transferring* them into string representation needed for export. Note that there is no stage in which nodes are deleted. The reason for this is the fact that the syntax tree is created by the parser, which only adds nodes to the tree during the parsing. The nodes cannot be modified outside the context of the parser, otherwise it could result in syntactical inconsistency.

3.2.4.3 Symbol Table

The *SymbolTable* becomes *non-empty* when it is initialized with standard records. Next its state can change through *adding, deleting,* and *retrieving* of records. The last possible state in which the symbol table can pass through is *local semantic checking,* during which local symbols are resolved, and checked for semantic correctness. The state transition diagram for symbol table is in Figure 3-17.

3.2.4.4 Parser

The parser is another interesting object whose state transition diagram is shown in Figure 3-18. What makes the parser interesting is the fact that it is one of the active objects that can be found in the EIS system. When it is created, it stays in an *idle* state until a request comes to parse an input stream, which transfers the parser into the *parsing* state in which parsing itself is performed. There are two other states through which the

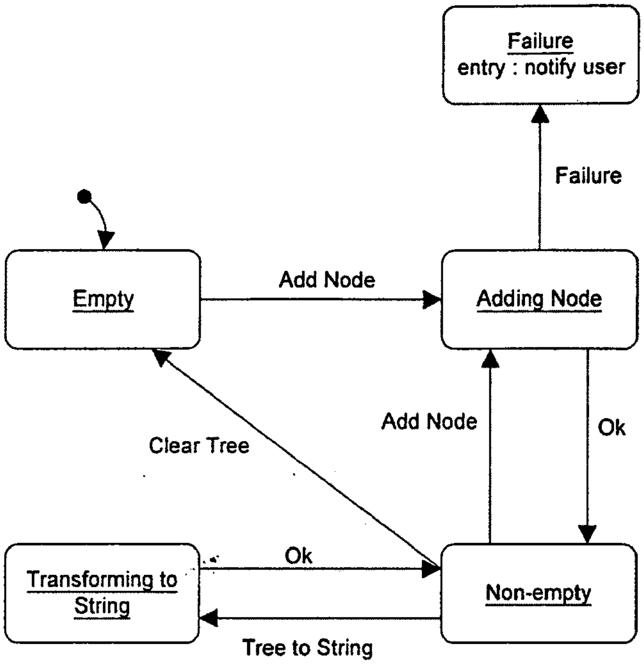

Figure 3-16
State Transition Diagram for Syntax Tree

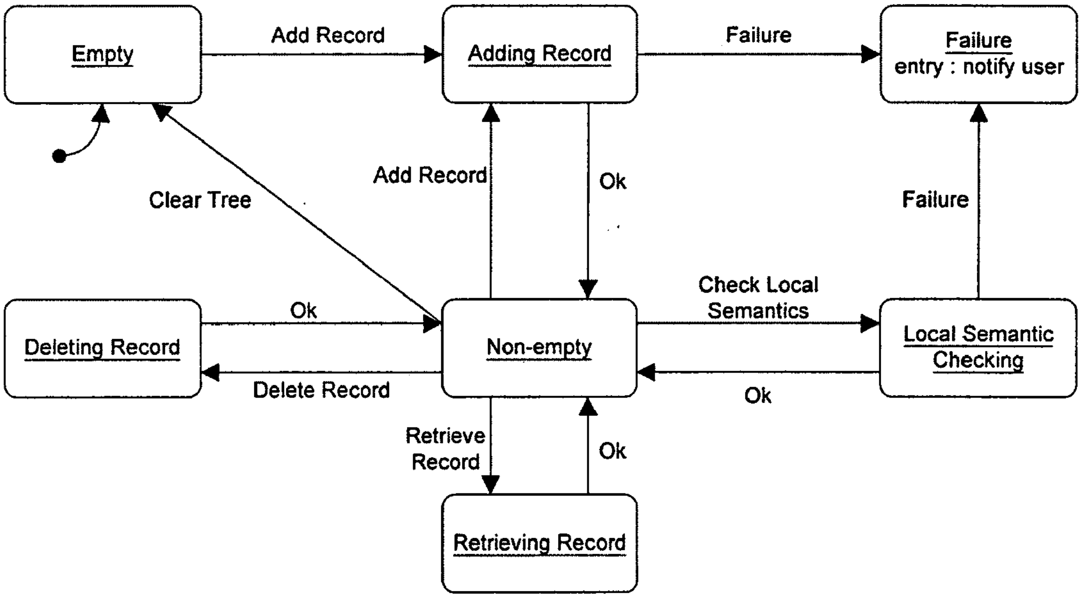

Figure 3-17
State Transition Diagram for Symbol Table

parser goes during the parsing stage, *creating the syntax tree*, and *creating the symbol table*. If an error occurs, *failure* is reported and parser returns to *idle* state.

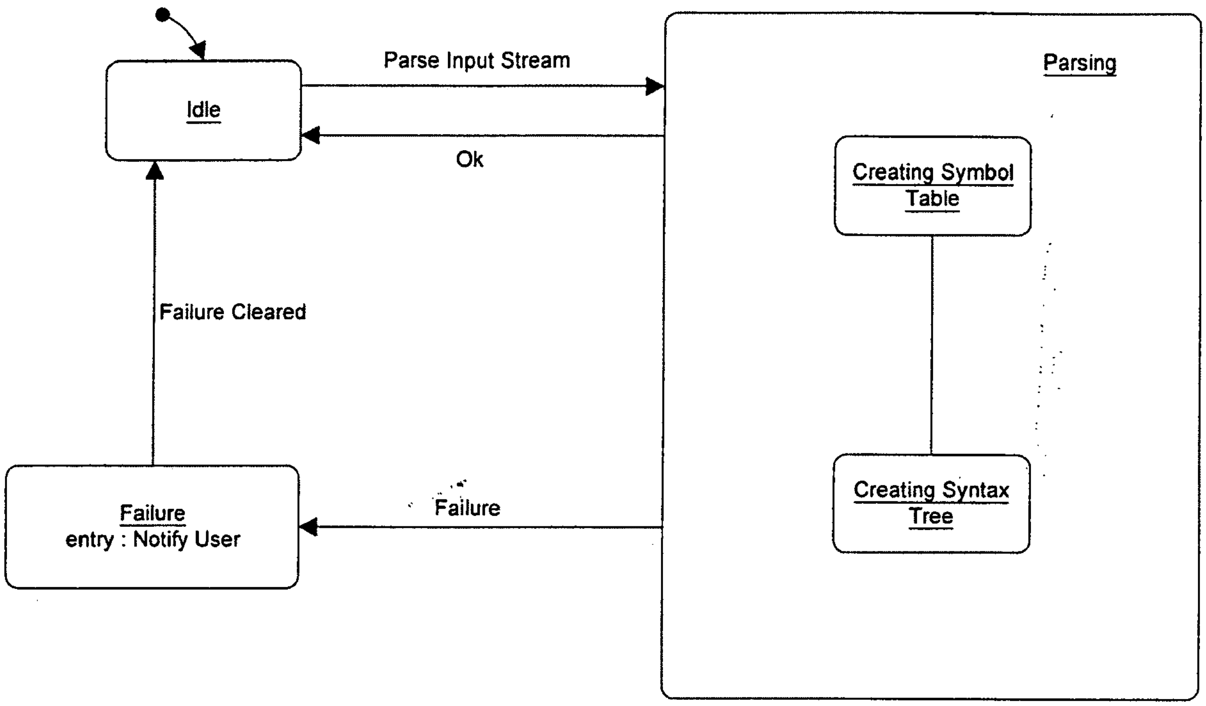

Figure 3-18
State Transition Diagram for Parser

3.2.4.5 Graphical User Interface

The graphical user interface (GUI) is another active object in the system. After it is initialized, it waits for user input. When the input arrives, it *handles user request* accordingly. In some cases, for example when user adds or deletes an object from the database, it triggers a change to *update* the display. The state transition diagram for the graphical user interface is shown in Figure 3-19.

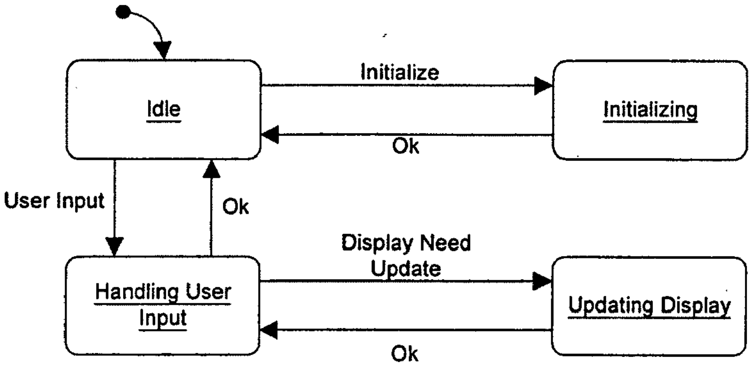

Figure 3-19
State Transition Diagram of Graphical User Interface (GUI)

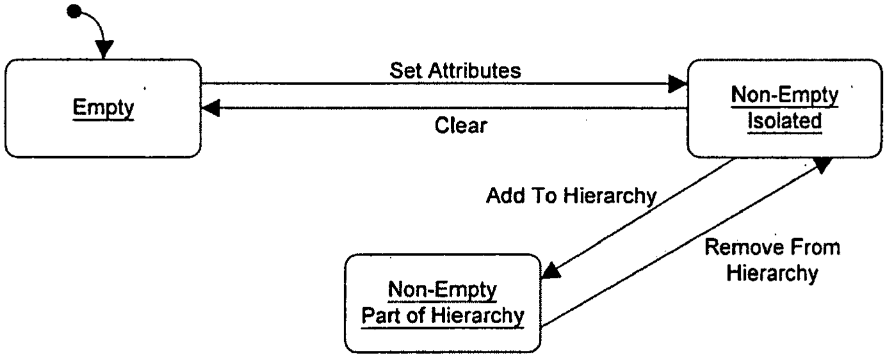

Figure 3-20
State Transition Diagram of EIS Hierarchy Node

3.2.4.6 EIS Hierarchy Node

Figure 3-20 shows the three possible states in which EIS Hierarchy Node can occur. At the beginning it is *empty*, subsequently it is *filled*, and finally it is *added* to the hierarchy.

3.2.4.7 Node Description

Node description is the last important object to be described. The key states are *empty*, *non-empty*, *adding attributes*, and *converting to stream*. The last state is very important, because it converts the node description that has been filled in by the user to an input stream that is used as an input into the parser. The state transition diagram for node description is in Figure 3-21.

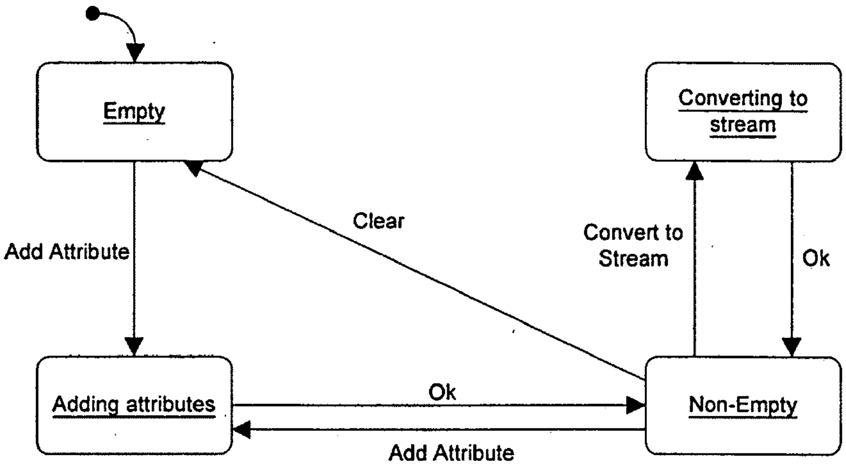

Figure 3-21
State Transition Diagram of Node Description

3.2.5 Interaction Diagrams of Key Scenarios

3.2.5.1 Creating New EIS Hierarchy Node

The interaction diagram in Figure 3-22 shows the process of creating a new node of the EIS hierarchy. The entire process starts at *EisGui* object. It first queries its own display to find out which node is currently selected - the current node will become parent of the new EIS node. The name of the parent node is passed to *NodeDescription* object. Next, the user fills out the description of the new node through series of dialogs, which are not shown in this diagram in order to make it more readable. The exact dialog is not of particular importance in overall system operation, yet it can be very important to fine tune the interface to meet the user's expectations [8].

At this point there exists a *NodeDescription* object that contains the description of the new EIS node. Since the parser operates on streams, the NodeDescription object converts its own representation into a stream which is subsequently forwarded to the *Parser* object. The *Parser* processes the stream token by token. During this process it fills up both the *SyntaxTree* and the *SymbolTable* objects. If a syntax error occurs during the parsing, the *Parser* creates a *ParseError* object which enables the system to propagate the error message to the user through the *EisGui*. If the *Parser* processes the entire stream without an error, control is returned to the *EisGui*, along with the symbol table and the syntax tree. *EisGui* sets additional attributes of the *EisHierNode* object, and finally sets the newly created symbol table and syntax tree to the *EisHierNode* object to complete the creation process.

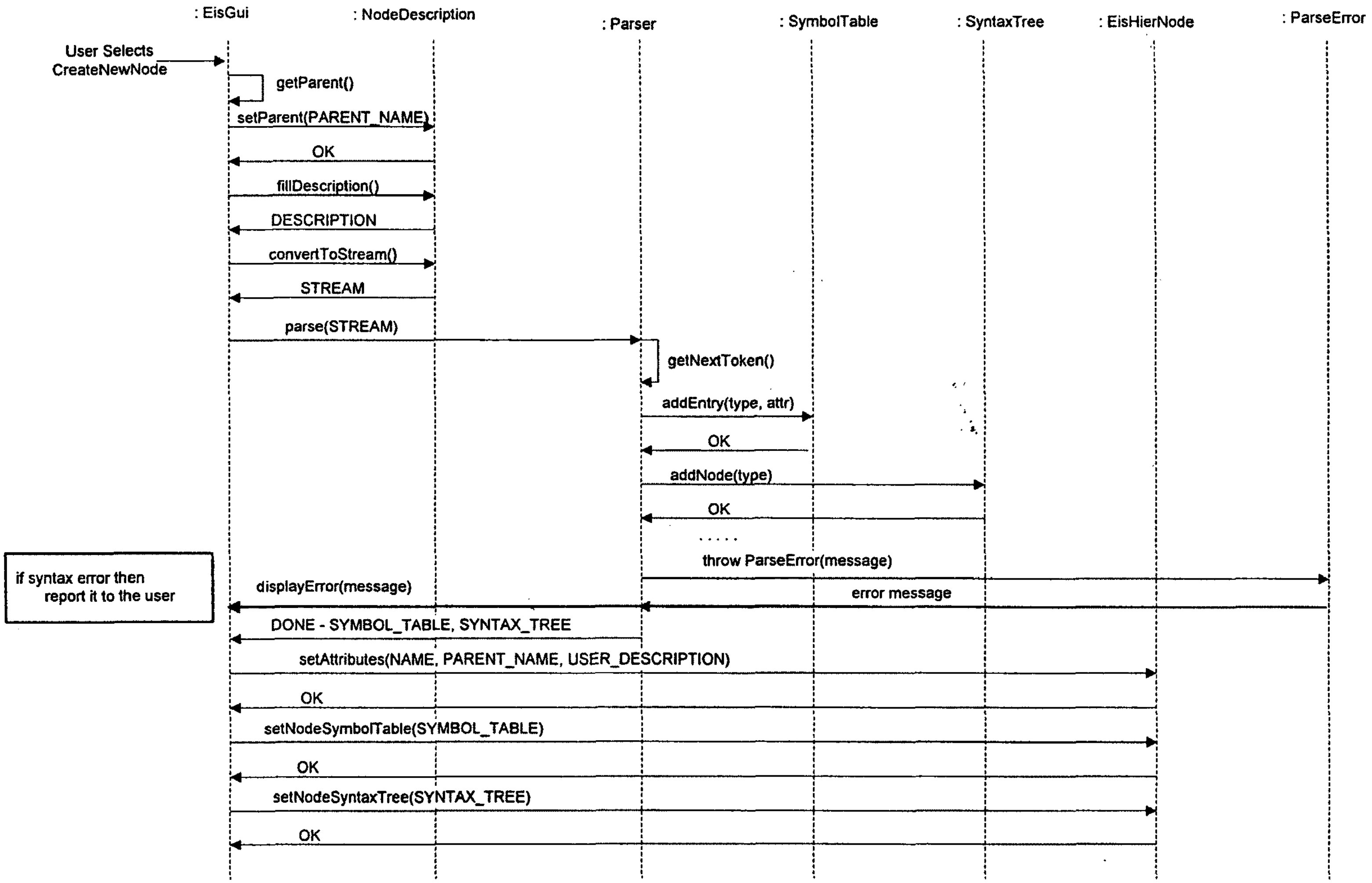

Figure 3-22
Interaction Diagram: Create New Object

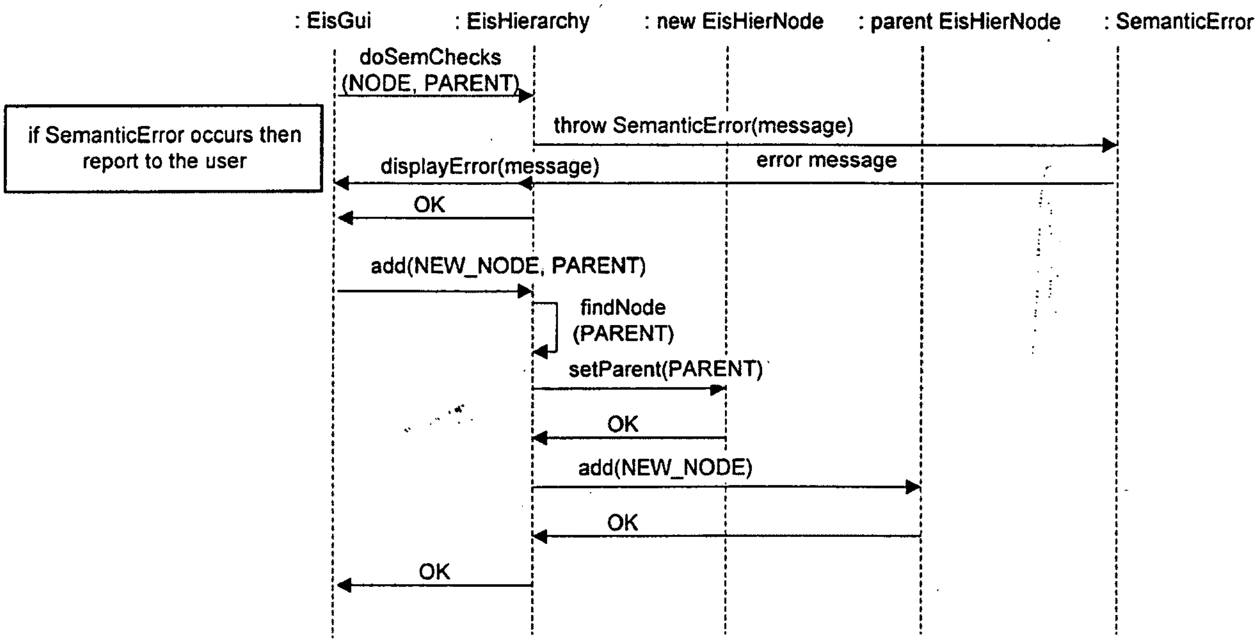

Figure 3-23
Interaction Diagram: Adding Node to EIS Hierarchy

3.2.5.2 Adding New Node to the EIS Hierarchy

The previous section explained the process of creating a new EIS node. Figure 3-23 depicts the process of adding this newly created node to the hierarchy. First, semantic checks are performed in the context of the hierarchy. If a semantic error occurs, *EisHierarchy* creates a *SemanticError* object that propagates back to the *EisGui* to be displayed to the user. If there are no semantic errors, a new *EisHierNode* is passed to the *EisHierarchy* for addition as a child of the previously determined current (parent) node.

3.2.5.3 Modifying Existing Node of the EIS Hierarchy

Figure 3-24 captures the process of modifying existing node in the EIS hierarchy. First *EisGui* identifies the name of the node that user wants to modify based on the selection in its own display. Next, *EisHierarchy* is asked for the description of the node. The *Description* is filled based on the information in the *SymbolTable* object of the *EisHierNode* that user wishes to modify. After the *Description* is filled, control returns back to the *EisHierarchy*, then to the *EisGui*. After the *Description* is modified by the user in the user interface, the object is processed by the *Parser* object in the same fashion as in the scenario in Figure 3-22. This processing is necessary, because both syntactic and semantic consistency must be checked following any modification. Some of the details of the parsing process are omitted, because they are identical to the ones in Figure 3-22. After the processing completes successfully, the newly created symbol table and syntax tree objects are returned and subsequently added to the modified *EisHierNode*. Finally, to check for global semantic consistency, semantic checks are performed on the modified node in the context of the *EisHierarchy*. On successful completion the node is modified within the hierarchy.

3.2.5.4 Saving EIS Hierarchy

The process of saving of an EIS hierarchy is a very simple one. First, the user selects a file where he/she wants the hierarchy to be saved. Next, *EisGui* sets the filename attribute in the *EisHierarchy* object, and also sets the hierarchy as saved by setting the appropriate boolean attribute of the *EisHierarchy* object. In the last step, the entire *EisHierarchy* objects is passed to the *EisHierarchyIO* object which performs the file output operation. The state transition diagrams of the saving process is shown in Figure 3-25.

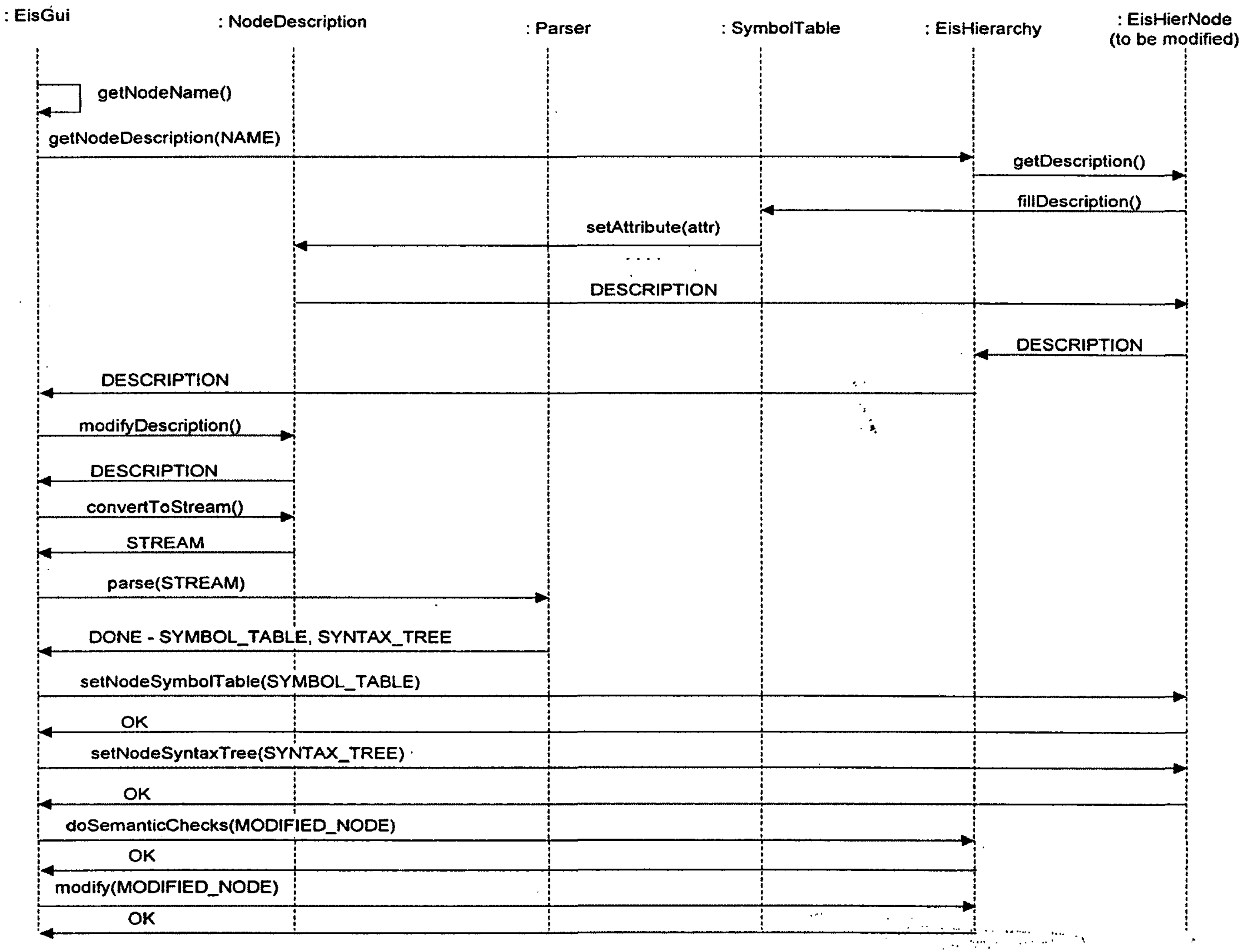

Figure 3-24
Interaction Diagram: Modifying Existing EIS Hierarchy Node

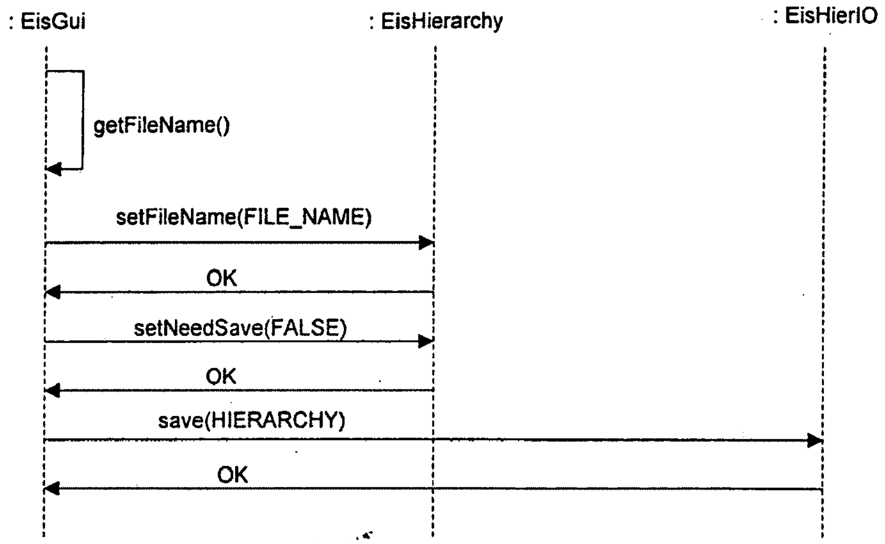

Figure 3-25
Interaction Diagram: Saving EIS Hierarchy

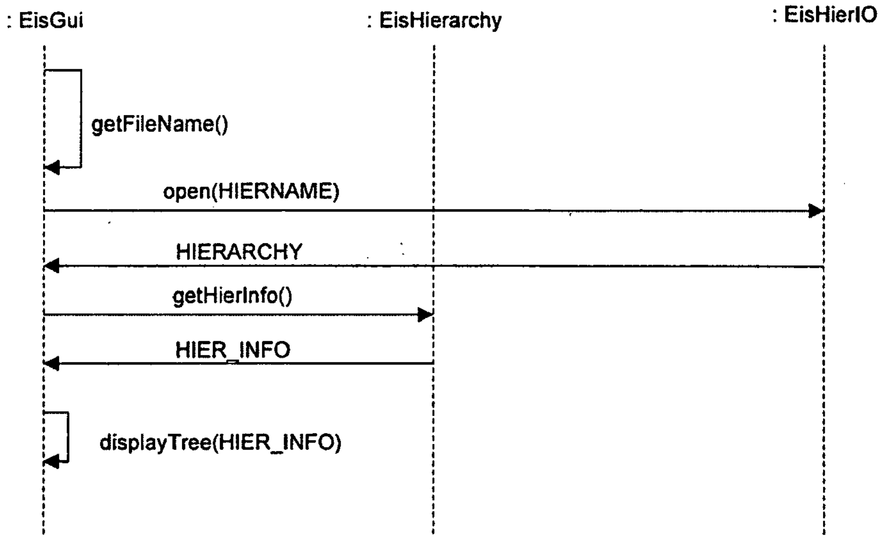

Figure 3-26
Interaction Diagram: Opening EIS Hierarchy

3.2.5.5 Opening Existing EIS Hierarchy

The process of opening an existing EIS hierarchy is the dual process to the save function. First, the user decides which hierarchy to open through the input to the *EisGui*. Next, *EisHierIO* object retrieves the requested EIS hierarchy, and returns it back to the *EisGui*. Finally, the *EisGui* displays the hierarchy structure based on information obtained from the *EisHierarchy* object. This scenario is shown in Figure 3-26.

3.2.5.6 Exporting EIS Hierarchy

When the EIS hierarchy is exported, its textual description is saved to an external file. This is in contrast to the save process, in which the entire EIS hierarchy object (including the symbol table and the syntax tree) is saved in binary form. For export, the user is asked by the *EisGui* for the name of the file into which the EIS hierarchy is to be exported and for the type of export (text or HTML). Next, the *EisGui* requests the *EisHierarchy* object to return its textual (or HTML) representation. There is a significant difference between the process of transforming the hierarchy to text format and transforming it to HTML format. The textual description is obtained directly from the *SyntaxTree* object of each *EisHierNode* via a syntax tree traversal; the resulting file should be syntactically correct description of the hierarchy if later imported. On the other hand, the HTML format does not necessary have to conform to EIS language specification, but rather it has to be formatted for the use in a Web browser. For this reason, *SymbolTable* object is used to create the HTML description, because it has easier access to the symbols of the hierarchy. During this process, all the nodes of the EIS hierarchy are converted into the appropriate format, but for simplicity there is only one instance of the *EisHierNode* used in the interaction diagram in Figure 3-27. Finally, the node description propagates up to the EisGui, and is passed to the *EisHierIO* object which outputs it into a file.

3.2.5.7 Importing EIS Hierarchy

The last scenario described in this chapter is the importing of an EIS hierarchy. When a hierarchy is imported, its textual description is separated into individual nodes by the *EisHierIO* object. These nodes are being parsed in manner previously described by the *Parser*, returning the syntax tree and symbol table objects. Next, a new *EisHierNode* object is created using the generated syntax tree and symbol table, and added to the *EisHierarchy*. The main difference between this process and adding a user created node to the EIS hierarchy is in the semantic checks. When a new node is created by the user through the user interface, semantic checks are performed before the node is actually added to prevent a semantic inconsistency being introduced into the hierarchy. However, when the hierarchy is being imported, we cannot perform the semantic checks one node at a time, because one node may depend on other object in the hierarchy that has yet to be imported. For this reason, the hierarchy is first populated with nodes, and then is validated by performing semantic checks on all the nodes in the hierarchy context. If a semantic error occurs, the whole hierarchy is discarded and user is informed about the nature of the error. An interaction diagram showing the process of importing EIS hierarchy is depicted in Figure 3-28.

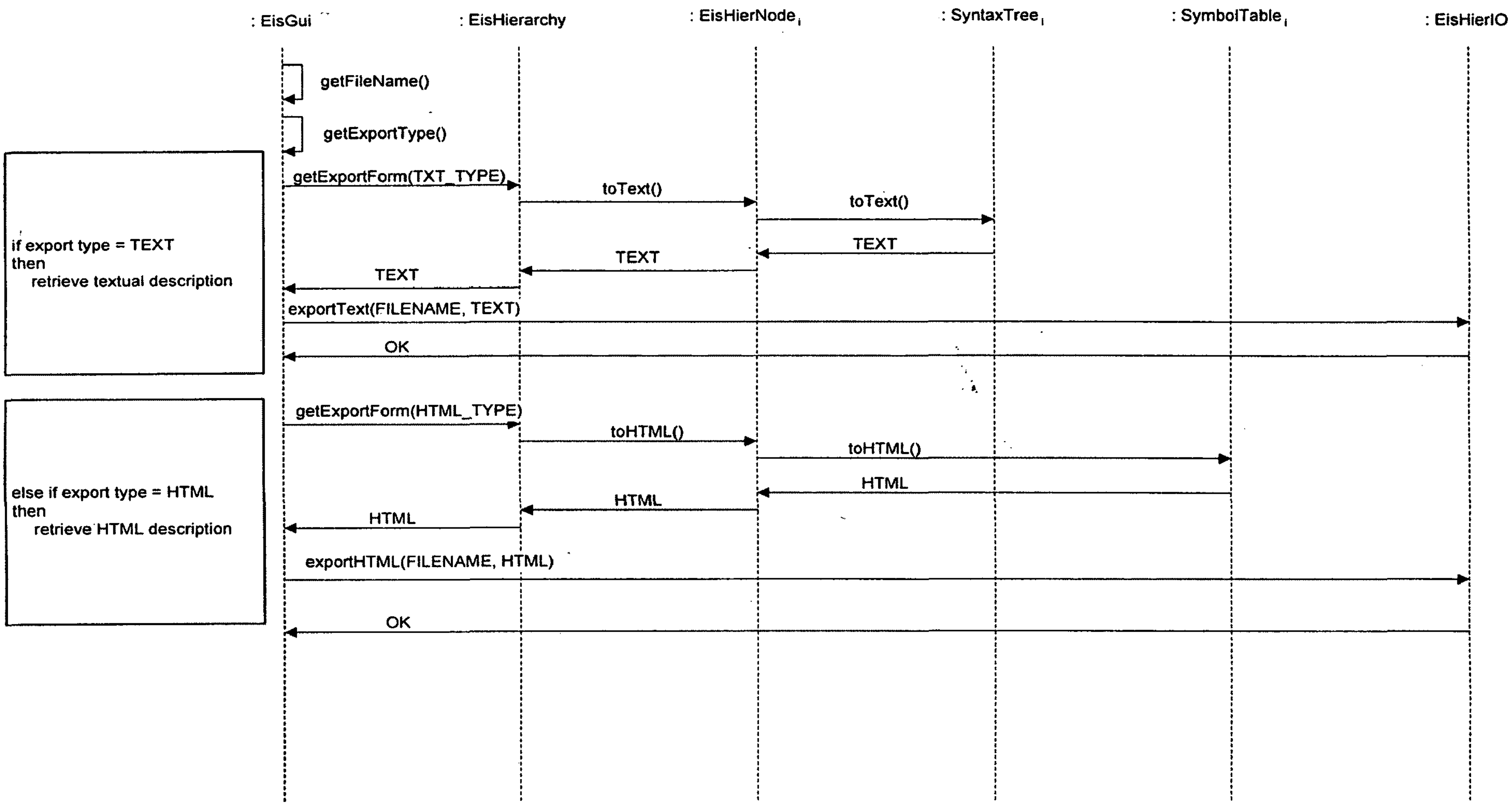

Figure 3-27
Interaction Diagram: Hierarchy Export

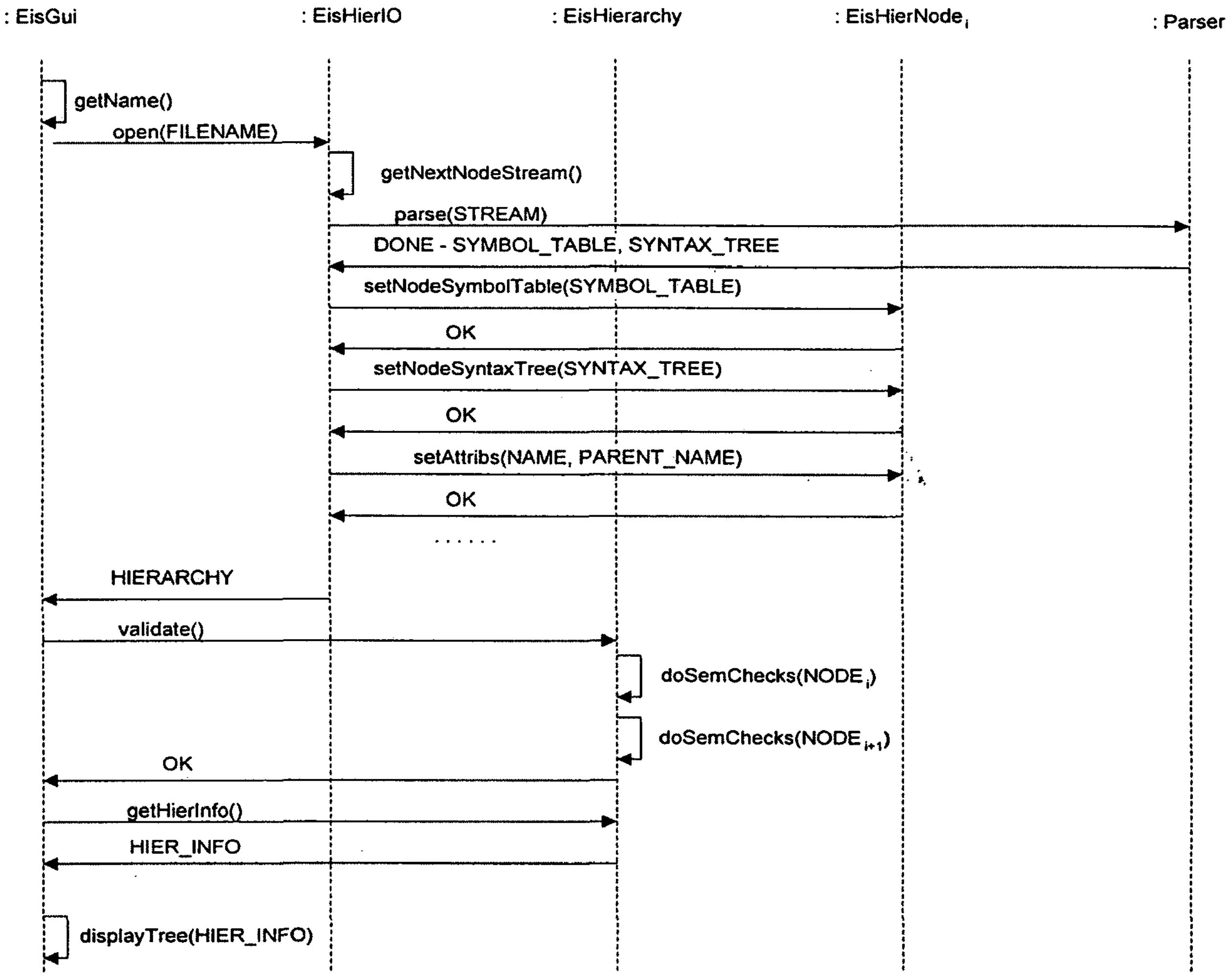

Figure 3-28
Interaction Diagram: EIS Hierarchy Import

Chapter 4

Implementation

4.1 Implementation Language : Why Java ?

When deciding on the implementation details of this project, the first thing that had to be decided was the implementation language. The two main competing choices were C++ and Java. C++ was the language of the original EIS, and so it could have been possible to reuse some parts of the code. However, in my opinion using C++ would not have solved many problems associated with the previous release of EIS. Java on the other hand, seemed to be a promising new language that many people thought of as the "next big thing", but at the time this project started there were many questions as to whether or not Java would really deliver what it promised. Therefore, I tested Java thoroughly before deciding to go forward with the implementation in this language. This part of my work can be thought of as a risk assessment phase of the spiral model, followed by prototyping effort.

There were several specific tasks that I tested during the Java evaluation phase of the development effort, including the following.

- Is Java really platform and architecture neutral?
- Does its interpreted nature have a negative effect on the performance?
- How effectively does the Java Virtual Machine (VM) deal with resource management?
- How robust is the Java VM?
- What is the learning curve for a C/C++ programmer to become proficient in Java?
- How easily is it integrated into a WWW?

These were the questions, and here I present the answers that I have found during my research.

I had the opportunity to test my Java programs on many different platforms and achieved good results. The platforms that were available to me during this process were:

- RS6000 with AIX4.1
- PentiumPRO 200 with Windows NT4.0
- Pentium 133 with Windows 95
- SGI Octane with IRIX6.4

Since Java is an interpreted language, there were a lot of questions about whether its performance is suitable for high performance computing. I designed several tests, then compared Java solutions with comparable C++ implementations. The results of two of these simple tests were indicative of the results in general. The first test is a simple program that indirectly calls a locally defined function 10^9 times. The results of this test are rather mixed. If only a Java interpreter is used, the program's execution time is more than 10 times slower that the same C++ program. However, when Java interpretation is replaced by use of a Just-in-Time (JIT) compiler, which is available with most Java Development Kits (JDK) starting with version 1.1, the performance improves to a level almost comparable with C++, running only about 1.5 times slower. A second test involves implementation of a matrix addition operation, using very large matrices (200MB per matrix). This experiment confirms the results of the first test, that Java performance is roughly comparable to C++ when JIT is used.

The second experiment, however, shows an even more important aspect of Java, which is its dynamic resource management. During this test, memory was repeatedly allocated and deallocated, but the speed of the program is not much affected, showing that Java can deal efficiently with dynamic memory management. One of the strengths of Java over C++ is its built in facility for dynamic resource management and automatic garbage collection. This, while still not perfect, helps a great deal in the implementation of dynamic systems like EIS. Dynamic resource manipulation is the source of most errors in C++ programs, because the programmer has to do all resource management. Programming errors are very hard to trace and have negative impact on the robustness of the system. In Java, dynamic resource manipulation is done much more robustly, and thus the systems that are build using Java can be made more robust with less programming effort.

Since the syntax of Java is very similar to the syntax of C++, the time to learn this new language should be relatively short for C++ programmers. Moreover, Java enforces the rules of object-oriented programming much more strictly than C++, making Java good language for the implementation of systems that were designed using object-oriented methods. Finally, Java provides direct support for integration into the World Wide Web (WWW). This is very important for the EIS development, because one of the long term goals has been its full integration into the Web. Unfortunately, the current implementation of EIS does not have a Web interface for reasons that will be described later in this chapter.

Based on the evaluation described above, I decided following this evaluation of Java to go forward with using it as the implementation language for the EIS.

4.2 Implementation Problems and Solutions

4.2.1 The EIS Parser

At the beginning of the project, there was an important implementation issue - the implementation of static semantic checking for the EIS language in a manner consistent with the formal checks defined in the EIS attribute grammar. Since this grammar defines the exact structure of the language used to define objects in an EIS hierarchy, the solution to this problem is crucial for the implementation phase. In the previous versions of EIS, language processing was implemented using lex and yacc. One possible solution to this problem was to keep and improve the lex and yacc implementation of the EIS grammar, and access it through the native method interface provided by Java. There were two main difficulties associated with this solution. The complexity of the implementation would increase considerably, but more importantly the availability of lex and yacc on Unix systems only would completely defeat one of the reasons for using Java - portability.

Fortunately, there is another solution to this problem. In fall of 1996, Sun Microsystems developed an automated parser generator written entirely in Java and producing Java source code. It first appeared under the name Jack, but later was changed to JavaCC (in tradition of yacc, Java Compiler Compiler). JavaCC provides the programmer with an interface that is much more intuitive than lex and yacc. One of the powerful features of this language processing tool is the means of propagating information up the parse tree. In yacc information is passed up the parse tree using global variables labeled "$$". In JavaCC all non-terminals are implemented using functions, which can be used to return values up the parse tree to the calling object. An example grammar used to illustrate JavaCC processing is shown in Appendix B.

There is another important feature that distinguishes JavaCC from yacc. While the parser generated by yacc is a bottom-up LALR parser, JavaCC generates a top-down

LL(k) parser[9]. This has an impact on the type of grammars that each parser can handle. Top-down parsing can be viewed as attempt to find the leftmost derivation of the input string, which means that it generates the parse tree in pre-order fashion starting at the root. Because of this parsing strategy, top-down parsers cannot handle left-recursive grammars, i.e. grammars in which there is a derivation A => A*a* for some string *a*. The grammar that was used to enforce the correct syntax of the EIS objects includes left-recursive productions. This was not a problem in the previous versions of the system, because yacc generates a bottom-up parser which handles left-recursive grammars without problems. We can solve this problem by eliminating left recursion from the grammar as shown in[9]. The modified grammar is shown in Appendix C.

4.2.2 Java 1.1 Conversion

One of the problems that was not possible to anticipate at the beginning of the development process and during the risk analysis, was the transition between different versions of Java. In early 1997 Sun released the new Java 1.1 version. There have previously been several different releases of Java 1.0, but none of them have brought such dramatic changes into the API (Application Programming Interface) as Java 1.1. Apart from several cosmetic changes, such as changes to names of some methods in the API classes, Java 1.1 included a complete redesign of the event model. This change causes many problems with an existing Java-based user interface. The event model in Java is concerned with the handling of events that are generated through the java.awt libraries, such as pushing buttons, moving the mouse, or hitting a key. When Java 1.1 was released, almost the entire EIS graphical user interface was completed. There were two possible courses of action from this point on: to continue with the Java 1.0 implementation, or to change the entire system to version 1.1. I chose to convert the EIS implementation to Java 1.1, mainly to make it more up-to-date. Adapting to the change in the event model thus caused a big setback in the implementation timetable. However, this seems to be a good choice, because it appears that Java 1.1 will become the industry standard. While my decision may show better in the long run, it was not the best possible short term solution for two main reasons. First, it caused a major setback in the EIS implementation, since I had to re-write most of the event handling functionality that was already present at the EIS system when Java 1.1 was released. Second, I overestimated the speed with which Java 1.1 would become widely available. As a result, the current implementation of EIS is a standalone application, without a Web interface, because there is currently no Internet browser that supports Java 1.1.

4.2.3 Input/Output

One of the big implementation problems in the previous version of EIS was the implementation of the Input/Output (I/O) functions. The I/O structure used to allow hierarchies to be saved/restored to/from disk storage was very complex and unstable, and it has caused various runtime problems. Java solves this implementation issue by providing an API for high level I/O as a part of its java.io library. This interface provides object level I/O, meaning that I/O can be performed directly on class instances independently of their internal structure. This simplifies greatly the I/O for the EIS system, in which the programmer had to laboriously extract the components of objects of EIS hierarchy, which are complex nested objects containing nodes, symbol tables, syntax trees and other attributes. The only problem with the Java solution is in adapting to EIS object changes, i.e. if the source code changes and is re-compiled, it is no longer possible to restore hierarchy objects that were saved with the old object definitions. However, in this case it is possible to export the EIS hierarchy in textual, rather than object, form using the old version, then import it as text, reparse it, and save it using the new version of EIS.

4.3 Implementation and Distribution Structure

Java "packages" are sets of classes that are logically grouped together. By using packages, the implementation structure can be divided into smaller, self-contained units to form the EIS library. These packages are:

- eis.beans : defines several generic components used by the GUI
- eis.desc : defines the description of the nodes in the EIS hierarchy
- eis.gui : defines the Graphical User Interface of the EIS system
- eis.parser : defines the parser of the EIS system
- eis.symbol_table : defines the symbol table for EIS hierarchy nodes
- eis.syntax_tree : defines the syntax tree for the EIS hierarchy nodes
- eis.util : defines several utilities that are used by the other packages

This structure implements the notation of "class category" described in Chapter 3 and depicted in Figure 3-9. In addition to these packages, the system includes a simple driver object which is used to start up the system by instantiating the EisGui.

The whole EIS implementation is designed to be distributed using the *jar* utility provided by the new JDK1.1. This utility can archive the entire implementation into a single file which can be distributed as a library. Since *jar* is part of Java, it provides a platform independent distribution solution. Also with the distribution comes a simple installation script that creates the database directory, and sets the appropriate environment variables used by the system.

4.4 Conclusions and Status

The goals of this project set in Chapter 1 were completed, including the complete redesign of the EIS system and its implementation. The new version, EIS 3.0 solves many problems of the previous versions by providing more robust and portable system that pays attention to dynamic resource management, a more secure system, and more consistent and complete language processing. Of the principle goals set at the beginning of this project, only that of providing a Web interface was not completed. This failure is temporary due to the lack of support for Java 1.1 in current Internet browsers. I expect the this deficiency can be corrected easily when this support is available.

Appendix A - Class Specifications

Name: *AbstractNodeDesc*

Definition: Superclass of the main description classes.
Contains common attributes and methods.

Attributes:

name	:	String
parent_name	:	String
description	:	String
doc_list	:	Vector
keywd_list	:	Vector
empty	:	boolean

Methods:

setName(String name)	:	boolean
getName()	:	String
setParentName(String name)	:	boolean
getParentName()	:	String
setDescription(String desc)	:	boolean
getDescription()	:	String
addDocument(String name, String location)	:	boolean
getDocuments()	:	Vector
addKeyword(String keyword)	:	boolean
getKeywords()	:	Vector
setEmpty(boolean empty)	:	boolean
isEmpty()	:	boolean

Name: *EisHierNode*

Definition: Building block of EIS hierarchy structure.

Attributes:

nodeName	String
nodeDescription	String
nodeType	int
nodeDepth	int
nodeEmpty	boolean
nodeParent	EisHierNode
nodeChildren	Vector
nodeSyntaxTree	SyntaxTree
nodeSymbolTable	SymbolTable
nodeDocuments	Vector
nodeKeywords	Vector

Methods:

EisHierNode()	
assignTo(EisHierNode node)	boolean
setNodeName(String name)	boolean
getNodeName()	String
setNodeDescription(String desc)	boolean
getNodeDescription()	String
setNodeType(int type)	boolean
getNodeType()	int
addNodeDocument(EisDocumentDesc desc)	boolean
getNodeDocuments()	Vector
addNodeKeyword(EisKeywordDesc desc)	boolean
getNodeKeywords()	Vector
setNodeDepth(int depth)	boolean
getNodeDepth()	int
setNodeSymbolTable(SymbolTable st)	boolean
getNodeSymbolTable()	SymbolTable
setNodeSyntaxTree(SyntaxTree st)	boolean
getNodeSyntaxTree()	SyntaxTree
setEmpty(boolean empty)	boolean
isEmpty()	boolean
clear()	boolean
setParent(EisHierNode parent)	boolean
getParent()	EisHierNode
addChild(EisHierNode child)	boolean
getChildAt(int index)	EisHierNode
getNumChildren()	int
deleteChild(int index)	boolean

deleteChild(String name)	:	boolean
isEqual(EisHierNode node)	:	boolean
isRoot()	:	boolean
isLeaf()	:	boolean
toString()	:	String
print()	:	boolean

Name: *EisHierarchy*

Definition: Main object of the EIS system. Contains syntactical and semantic description of the system.

Attributes:	*rootNode*	:	EisHierNode
	currentNode	:	EisHierNode
	numNodes	:	int
	hierEmpty	:	boolean
	hierName	:	String
	hierCreator	:	String
	hierDescription	:	String
	dateCreated	:	Date
	fileName	:	File
	saved	:	boolean

Methods:	*EisHierarchy*()		
	newHier()	:	boolean
	setRoot(EisHierNode root)	:	boolean
	getRoot()	:	EisHierNode
	setEmpty(boolean empty)	:	boolean
	isEmpty()	:	boolean
	needSave(boolean save)	:	boolean
	isSaved()	:	boolean
	hasFile()	:	boolean
	setFile(File name)	:	boolean
	getFile()	:	File
	setHierName(String name)	:	boolean
	getHierName()	:	String
	setHierDescription(String desc)	:	boolean
	getHierDescription()	:	String
	getHierCreator()	:	String
	getHierDate()	:	String
	getCurrentNode()	:	EisHierNode
	setCurrentNode(EisHIerNode node)	:	boolean
	addChild(EisHIerNode node, String parent)	:	boolean
	deleteNode(String name)	:	boolean
	getUnboundParamList(String startNode)	:	Vector
	getUnboundVariableList(String startNode)	:	Vector
	doSemanticChecks(EisHierNode node, String parent)	:	boolean
	toString()	:	String
	print()	:	boolean

Name: *EisHierIO*

Definition: Defines methods for interaction between hierarchy objects and the filesystem.

Attributes:	*out*	:	ObjectOutputStream
	in	:	ObjectInputStream
	file	:	File

Methods:	*save*(EisHierarchy hier, File file)	:	boolean
	open(File file)	:	EisHierarchy
	import(File file)	:	EisHierarchy
	importNode(File file)	:	EisHierNode
	exportText(EisHierarchy hier, File file)	:	boolean
	exportHTML(EisHierarchy hier, File file)	:	boolean
	exportNodeText(EisHierNode node, File file)	:	boolean
	exportNodeHTML(EisHierNode node, File file)		boolean

Name: *SymbolTableRecord*

Definition: Defines single entry in the symbol table structure.

Attributes:	*tag*	:	String
	entryType	:	int
	typeDenoter	:	int
	arguments	:	Vector
	retType	:	int
	constValue	:	String
	paramType	:	String
	arrayIndexList	:	Vector
	recordFieldList	:	Vector
	entryStatus	:	int

Methods:	*SymbolTableRecord*()		
	setTag(String tag)	:	boolean
	getTag()	:	String
	setEntryType(int type)	:	boolean
	getEntryType()	:	int
	setTypeDenoter(int typeDenoter)	:	boolean
	getTypeDenoter()	:	int
	addArgument(int arg)	:	boolean
	getArguments()	:	Vector
	setRetType(int retType)	:	boolean
	getRetType()	:	int
	setConstValue(String constValue)	:	boolean
	getConstValue()	:	String
	setParamType(String paramType)	:	boolean
	getParamType()	:	String
	addArrayIndex(String lower, String upper)	:	boolean
	getArrayIndexList()	:	Vector
	addRecordFieldId(int id)	:	boolean
	getRecordFieldList()	:	Vector
	setStatus(int status)	:	boolean
	getStatus()	:	int
	toString()	:	String
	print()	:	boolean

Name: *SymbolTable*

Definition: Stores symbols defined in a context of each node in hierarchy.

Attributes:	*symbolTable*		Vector
Methods:	*SymbolTable*()		
	clear()	:	boolean
	getNumRecords()	:	int
	getRecord(int index)	:	SymbolTableRecord
	addRecord(SymbolTableRecord rec)	:	boolean
	removeRecord(int index)	:	boolean
	lookup(String id)	:	int
	lookupAdd(String id)	:	int
	getType(String id)	:	int
	addSetType(String typeName)	:	boolean
	addFwdDeclList(Vctor idList)	:	boolean
	addIntUsesList(Vector idList)	:	boolean
	addParamDecl(String id, String type)	:	boolean
	addParamBind(String id1, String id2)	:	boolean
	addTypeDefn(String name, int type)	:	boolean
	addVarDefn(Vector idList, int type)	:	boolean
	addConstantDefn(String id1, String id2, String value)	:	boolean
	addFuncDefn(String name, Vector arguments, String retValue)	:	boolean
	addArgDecl(int type)	:	boolean
	addArrayType(Vector indexList, int type)	:	boolean
	addRecordFieldType(Vector idList, int type)	:	boolean
	addRecordType(Vector fieldList)	:	boolean
	addStvarBind(String id, String value)	:	boolean
	getIntUsesList()	:	Vector
	getFwdDeclList()	:	Vector
	getParamDeclList()	:	Vector
	getBoundParamList()	:	Vector
	getVariableDeclList()	:	Vector
	getBoundVariableList()	:	Vector
	getTypeList()	:	Vector
	getConstList()	:	Vector
	getFunctionList()	:	Vector
	getTypeString(int record)	:	String
	toString()	:	String
	print()	:	boolean

Name: *SimpleNode*

Definition: Represents a single node in the syntax tree structure

Attributes:	*parent*	:	Node
	children	:	Vector
	identifier	:	String
	info	:	Object
	nodeType	:	int

Methods:	*SimpleNode*()		
	SimpleNode(String id, int type)		
	clear()	:	boolean
	jjtCreate(String id, int type)	:	Node
	jjtSetParent(Node parent)	:	boolean
	jjtGetParent()	:	Node
	jjtSetType(int type)	:	boolean
	jjtGetType()	:	int
	jjtAddChild(Node node)	:	boolean
	jjtGetChild(int index)	:	Node
	jjtGetNumChildren()	:	int
	setInfo(Object info)	:	boolean
	getInfo()	:	Object
	toString()	:	String
	print()	:	boolean

Name: *SyntaxTree*

Definition: Defines the syntax of each node in the EIS hierarchy

Attributes:

rootNode	:	SimpleNode
empty	:	boolean

Methods:

SyntaxTree()		
SyntaxTree(SimpleNode root)		
clear()	:	boolean
assignTree(SimpleNode root)	:	boolean
setEmpty(boolean empty)	:	boolean
isEmpty()	:	boolean
toString()	:	String
print()	:	boolean

Appendix B - JavaCC Example

Grammar:	E	->	T E’
	E’	->	+ T E’ | ε
	T	->	F T’
	T’	->	* F T’ | ε
	F	->	(E) | **id**

JavaCC implementation:

```
PARSER_BEGIN(TestParser)
public class TestParser
{
        public static void main(String[] args) throws ParseError
        {
                TestParser parser = new TestParser(System.in);
                parser.E();
        }
}
PARSER_END(TestParser)

IGNORE_IN_BNF:
{}
{
        “ “
        | “\t”
        | “\n”
}

TOKEN:
{}
{
        <PLUS :                “+” >
        | <MULTIPLY :          “*” >
        | <LEFT_PAREN :        “(“ >
        | <RIGHT_PAREN :  “)” >
        | < ID : [“a” - “z”, “A” - “Z”] ([“a” - “z”, “A” - “Z”, “0” - “9”])* >
}
```

```
void E() :
{}
{
        T() E_prime() <EOF>
}

void E_prime() :
{}
{
        ( <PLUS> T() E_prime() )*
}

void T() :
{}
{
        F() T_prime()
}

void T_prime() :
{}
{
        ( <MULTIPLY> F() T_prime() )*
}

void F() :
{}
{
        <LEFT_PAREN> E() <RIGHT_PAREN>
        | <ID>
}
```

Appendix C - BNF for EisParser

```
CombinedSyntax      ::= (    ClassDefn
                             | InstanceDefn
                             | MethodDefn  )*
                     <EOF>

ClassDefn           ::= <CLASS>
                             <IDENTIFIER>
                             <OF>
                              <IDENTIFIER>
                             InterfaceUses
                             ForwardDeclaration
                             BindParameter
                             ParameterDeclaration
                             Description
                             MixedDeclarationList
                             BindStateVariables
                             Keywords
                             Document
                    <END_CLASS>

InstanceDefn        ::= <INSTANCE>
                             <IDENTIFIER>
                              <OF>
                              <IDENTIFIER>
                             BindParameter
                             Description
                             BindStateVariables
                             Keywords
                             Documents
                    <END_INSTANCE>
```

```
MethodDefn               ::= <METHOD>
                                 <IDENTIFIER>
                                  <OF>
                                  <IDENTIFIER>
                                 <TO>
                                 <IDENTIFIER>
                                 Path
                                 Description
                                 Keywords
                                 Documents
                             <END_METHOD>

Path                     ::= <PATH>
                                  <STRING_LITERAL>

ForwardDeclaration       ::= ( <FORWARD_DECL>
                                 IdentifierList
                             <END_FORWARD_DECL> )*

InterfaceUses            ::= ( <INTERFACE_USES>
                                 IdentifierList
                             <END_INTERFACE_USES> )*

ParameterDeclaration             ::= ( <PARAM_DECL>
                                         ParameterDeclarationList
                                     <END_PARAM_DECL> )*

ParameterDeclarationList         ::= <IDENTIFIER>
                                         <COLON>
                                         ParameterType
                                         ParameterDeclarationList_prime


ParameterDeclarationList_prime           ::= ( <SEMI>
                                                   <IDENTIFIER>
                                                  <COLON>
                                                  ParameterType
                                             ParameterDeclarationList_prime )*

ParameterType            ::= <CLASS>
                           | <TYPE>
                           | <CONST>
                           | <FUNCTION>
```

```
BindParameter            ::= ( <PARAM_BIND>
                                   BindParameterList
                             <END_PARAM_BIND> )*

BindParameterList        ::= <IDENTIFIER>
                                   <ASSIGNOP>
                                   <IDENTIFIER>
                                   BindParameterList_prime

BindParameterList_prime        ::= ( <SEMI>
                                         <IDENTIFIER>
                                         <ASSIGNOP>
                                          <IDENTIFIER>
                                   BindParameterList_prime )*

MixedDeclarationList           ::= ( MixedDeclaration
                                         <SEMI>
                                    MixedDeclarationList )*

MixedDeclaration         ::= TypeDefn
                           | VarDefn
                           | ConstantDefn
                           | FunctionDefn

BindStateVariables       ::= ( <STVAR_BIND>
                                   BindStateVariablesList
                           <END_STVAR_BIND> )*

BindStateVariablesList         ::= <IDENTIFIER>
                                         <ASSIGNOP>
                                         ValueOrId
                                         BindStateVariablesList_prime


BindStateVariablesList_prime         ::= ( <SEMI>
                                               <IDENTIFIER>
                                               <ASSIGNOP>
                                               ValueOrId
                                         BindStateVariablesList_prime )*

ValueOrId       ::= <IDENTIFIER>
                  | Value
```

```
TypeDefn      ::= <TYPE>
                   <IDENTIFIER>
                    <ASSIGNOP>
                   TypeDenoter

VarDefn       ::= <VAR>
                   IdentifierList
                   <OF>
                   TypeDenoter

ConstantDefn  ::= <CONST>
                   <IDENTIFIER>
                   <COLON>
                   <IDENTIFIER>
                   <ASSIGNOP>
                   Value

FunctionDefn  ::= <FUNCTION>
                   <IDENTIFIER>
                   <LEFT_PAREN>
                   ArgumentList
                   <RIGHT_PAREN>
                   <COLON>
                   <IDENTIFIER>

ArgumentList  ::= ArgumentDeclaration
                   ArgumentList_prime
                | ArgumentList_prime

ArgumentList_prime  ::= ( <COMMA>
                            ArgumentDeclaration
                      ArgumentList_prime )*

ArgumentDeclaration ::= TypeDenoter

TypeDenoter         ::= <IDENTIFIER>
                      | NewType

NewType             ::= ArrayType
                      | RecordType
                      | SetType

RecordType          ::= <RECORD_START>
                            FieldList
                      <RECORD_END>
```

```
FieldList              ::= RecordSection
                               FieldList_prime

FieldList_prime        ::= ( <SEMI>
                               RecordSection
                           FieldList_prime )*


RecordSection          ::= IdentifierList
                               <COLON>
                               TypeDenoter

ArrayType              ::= <ARRAY>
                               <LEFT_SQUARE_BR>
                               IndexTypeList
                               <RIGHT_SQUARE_BR>
                               <OF>
                               TypeDenoter

IndexTypeList          ::= IndexType
                               IndexTypeList_prime

IndexTypeList_prime ::= ( <COMMA>
                               IndexType
                           IndexTypeList_prime )*

IndexType              ::= LowerBound
                               <DOTDOT>
                               UpperBound

LowerBound             ::= Value
                           | <IDENTIFIER>

UpperBound             ::= Value
                           | <IDENTIFIER>

SetType                ::= <SET>
                               <OF>
                               BaseType

BaseType               ::= <IDENTIFIER>
```

```
Keywords                ::= ( <KEYWORDS>
                                KeywordsList
                            <END_KEYWORDS> )*

KeywordsList            ::= <STRING_LITERAL>
                                KeywordsList_prime

KeywordsList_prime  ::= ( <SEMI>
                                <STRING_LITERAL>
                            KeywordsList_prime )*

Documents               ::= ( <DOCUMENTS>
                                DocumentDefnList
                             <END_DOCUMENTS> )*

DocumentDefnList      ::= DocumentDefn
                                DocumentDefnList_prime

DocumentDefnList_prime      ::= ( <SEMI>
                                        DocumentDefn
                                    DocumentDefnList_prime )*

DocumentDefn                ::= <DOCUMENTNAMELOC>
                                        <IDENTIFIER>
                                        <STRING_LITERAL>
                                    | <DOCUMENTATION>
                                        <STRING_LITERAL>

Value           ::= <INTEGER_LITERAL>
                | <FLOATING_POINT_LITERAL>
                | <STRING_LITERAL>
                | <CHARACTER_LITERAL>
                | Boolean

IdentifierList  ::= <IDENTIFIER>
                        IdentifierList_prime

IdentifierList_prime   ::= ( <COMMA>
                                <IDENTIFIER>
                            IdentifierList_prime )*

Description             ::= <MULTI_LINE_STRING_LITERAL>
                            | <STRING_LITERAL>

Boolean                 ::= <TRUE> | <FALSE>
```

Bibliography

[1] Palaiya, Vijayant. *Design and construction of language processor based on attribute grammar for EIS*, MS Thesis, University of Montana, 1996

[2] Righter, Ronald. *An object-oriented analysis of ecosystem modeling*, MS Thesis, University of Montana, 1993

[3] Hemige, Venugopal. *A Qualitative and Quantitative comparison of the OSF DCE and the SUN RPC.*, MS Thesis, University of Montana, 1992

[4] R. Ford, R. Righter, M. Sweet, P. Votava. *A Web-based tool for data organization and dissemination*, University of Montana, 1997

[5] Booch, Grady. *Object-oriented analysis and design with applications 2nd ed.* The Benjamin/Cummings Publishing Company, Inc., 1994

[6] Boehm, B. *A Spiral Model of Software Development and Enhancement* Software Engineering Notes vol. 4(10), 1986

[7] Mondava, Srinivas. *Graphical User Interface for the EIS* MS Project, University of Montana, pending

[8] Reimer, Yolanda, *Design of Algorithm Animations* MS Thesis, University of Montana, 1996

[9] A. Aho, R. Sethi, J. Ullman. *Compilers - Principles, Techiques, and Tools* Addison-Wesley Publishing Company, 1988