A PROTOTYPICAL IMPLEMENTATION OF GALAHAD: A CONCEPTUAL MODELING LANGUAGE USING THE OBJECT PARADIGM

by

MARK McKAY NICKSON

B.S., Virginia Polytechnic Institute and State University, 1983

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Management Science and Information Systems

1988

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA-88-240
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT UNIVERSITY OF COLORADO / 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS *(City, State, and ZIP Code)*	7b. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)* (UNCLASSIFIED)
A PROTOTYPICAL IMPLEMNTATION OF GALAHAD:
A CONCEPTUAL MODELING LANGUAGE USING THE OBJECT PARADIGM

12. PERSONAL AUTHOR(S)
MARK McKAY NICKSON

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
THESIS/~~DISSERTATION~~	FROM ______ TO ______	1988	212

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b. TELEPHONE *(Include Area Code)* (513) 255-2259 / 22c. OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

This thesis for the Master of Science degree by

Mark McKay Nickson

has been approved for the

Department of

Management Science and Information Systems

by

[illegible]

David E. Monarchi

[illegible]

Robert H. Taylor

Date 12/6/88

Nickson, Mark McKay (M.S., Management Science/Information Systems)

A Prototypical Implementation of Galahad: A Conceptual Modeling Language Using the Object Paradigm

Thesis directed by Associate Professor David E. Monarchi

This thesis discusses a DOS micro computer implementation of Galahad, a conceptual modeling language based on the object paradigm. The language chosen for this particular implementation was PC Scheme, a dialect of LISP developed by Texas Instruments. The focus of the discussion is on the code for the implementation.

To place the implementation in perspective, a general discussion of the object paradigm is presented along with the specific object-oriented specifications required for the Galahad system. Also, a brief survey of the major object-oriented languages available today is presented.

The reader is assumed to have a basic understanding of the object paradigm and is also assumed to be intimately familiar with the Scheme programming language.

DEDICATION

To my parents, Marion and Ben Nickson

ACKNOWLEDGEMENTS

I first would like to thank Professors Monarchi and Taylor for their careful review of this paper and their invaluable editorial comments. I owe special thanks to Professor Monarchi for his assistance with the design of the prototype and his thoughtful insight in helping me tackle some tough coding problems.

I owe special gratitude to Professor Tegarden, for his continual patience in answering my seemingly endless stream of questions. Professor Tegarden has maintained a forest-level view of the prototype and has bailed me out of countless tree-level, and sometimes even knothole-level, coding problems. Professor Tegarden also deserves the initial credit for our analysis of the SCOOPS implementation.

Finally, I would like to thank my wife Pamela Ann. For the last two and a half years she has been eternally patient and supportive as I have pursued my Master's degree.

CONTENTS

CHAPTER

I. INTRODUCTION 1

The Object Paradigm 2

Object-Oriented Programming 3

Encapsulation 4

Information Hiding 4

Polymorphism 5

Data Abstraction 5

Class Abstraction Mechanisms 6

Inheritance 7

Dynamic Binding 7

II. THE GALAHAD LANGUAGE SPECIFICATION:
PRIMARY MODELING REQUIREMENTS 9

Conventional Object-Oriented Concepts 10

Lattice Inheritance 10

Instance Lattice Inheritance 12

Instance Anomalies 13

Abstract Superclasses 15

Simple Classes 17

Domain/Cardinality Constraint Enforcement 18

Specialized Methods 18

The Access-To Mechanism 22

Interclass Existence Dependencies 22

Instance Aggregation 23

Part Aggregation 25

Element Aggregation 26

Member Aggregation 26

Concluding Remarks on Instance Aggregation 27

III. SURVEY OF CURRENTLY AVAILABLE OBJECT-ORIENTED LANGUAGES 28

SIMULA 28

Smalltalk 30

ACTOR 31

FLAVORS 32

Loops 33

SCOOPS 34

CLOS 36

Comparison of Major Features of Current Object-Oriented Languages 37

IV. THE GALAHAD SYSTEM 44

Development Methodology 45

Coding Conventions 46

Galahad Basics 47

Galahad Primary Data Structures 47

Galahad Logical Design 48

Galahad Environment Frame Design 53

The Implementation of Message Passing in Galahad 57

Modeling Requirements Implementation 64

Lattice Inheritance 64

Instance Lattice Inheritance 66

Abstract Superclasses 71

Simple Classes 71

Domain/Cardinality Constraint Enforcement 74

Specialized Methods 76

Instance Aggregation 79

V. DIRECTIONS FOR FUTURE RESEARCH AND CONCLUSION 83

Minor Programming Tasks 83

Additional Functionality 88

Conclusion 95

BIBLIOGRAPHY 96

APPENDIX

A. A DESCRIPTION OF GALAHAD CLASS AND INSTANCE VECTORS 99

B. GALAHAD USER'S GUIDE 119

C. AN ANALYSIS OF TEXAS INSTRUMENT'S IMPLEMENTATION OF SCOOPS IN PC SCHEME 178

TABLES

Table

3-1 Comparison of Object-Oriented Languages 38

3-2 Glossary of Terms 40

FIGURES

Figure

2-1 Lattice Inheritance Example 11

2-2. Instance Lattice Inheritance Example 14

2-3 Instance Anomaly Example 14

2-4 Abstract Superclass Example 16

2-5 Domain/Cardinality Constraint Example 19

2-6 Specialized Methods Example 21

2-7 Instance Aggregation Example 24

4-1 Galahad Class and Instance Vectors 49

4-2 Galahad Logical Design 51

4-3 Initial Environment Frame Structure 54

4-4 Code Showing the Implementation of
Galahad's READ-EVAL-PRINT Loop 56

4-5 Galahad Environment Frame Structure 58

4-6 Implementation of Message Passing 60

4-7 Code Showing the Construction of
Environment Bindings 62

4-8 Environment Frame Structure During
Evaluation of a Method 63

4-9 Lattice Inheritance Example (Revisited) 65

4-10 Segment of COMPILE-CLASS Code
Used to Implement Lattice Inheritance 67

4-11 Instance Lattice Inheritance Example (Revisited) 68

4-12 Completed All-Instance-Variables Slot for the Instance DPT 70

4-13 Code Showing Implementation of Message Sending when an Instance has more than One Defining Class 72

4-14 Abstract uperclasses Example (Revisited) 73

4-15 %SET-GALAHAD% 75

4-16 Specialized Methods Example (Revisited) 77

-17 Code Showing the Implementation of Specialized Methods 78

4-18 Instance Aggregation Example (Revisited) 80

4-19 Instance Aggregation: Differentiation Among Different Subclasses 82

B-1 Proper Overall Format for a Galahad Application 122

C-1 SCOOPS Class Vector and Instance Environment Structure 180

C-2 SCOOPS and Lattice Inheritance for Methods 185

C-3 SCOOPS Environment Hierarchy: Multiple Classes with Multiple Instances 189

CHAPTER I

INTRODUCTION

The purpose of this paper is to describe the implementation of Galahad, a conceptual modeling language based on the object paradigm. The Galahad language specification was developed jointly by professors David E. Monarchi and David P. Tegarden. Professor Monarchi is an Associate Professor at the College of Business, University of Colorado, Boulder. Professor Tegarden is currently pursuing his doctoral degree from the same institution.

This work describes the implementation of a prototypical version of Galahad. We assume the reader is conversant in TI Scheme, a version of the Scheme programming language developed by Texas Instruments. Even though we provide a brief introduction to object-oriented concepts in the beginning of this paper, we presume the reader has some prior familiarity with the object paradigm.

The main body of this paper is designed to be read in its entirety. We begin our discussion by first introducing the reader to some of the basic tenants behind the object paradigm and object-oriented programming. Chapter 2 describes the Galahad language specification and its major modeling capabilities. Chapter 3 places the Galahad specification in the context of current literature by comparing Galahad to other object-oriented systems available today. Chapter 4 discusses the details behind the implementation of the language specification. Accompanying these details are segments of Galahad code that show how we implemented the language. Chapter 5 concludes this work by discussing the limitations of the current system and suggesting directions for future research.

Also accompanying this paper are several appendices which the reader can reference for additional information. Appendix A provides a comprehensive description of the data structures used to implement Galahad classes and instances. Appendix B serves as the Galahad user's guide. Appendix C describes our analysis of Texas Instrument's implementation of SCOOPS (SCheme Object Oriented Subsystem). We include the SCOOPS description in this thesis because our analysis of the implementation assisted us with the design of the Galahad system.

The Object Paradigm

The object paradigm can be viewed as an information modeling concept that describes conceptual entities as objects. The relationships among these conceptual entities are directly modeled as logical connections among the objects. Objects communicate their local state and request services of other objects by sending messages.

To define the term "object", we will conform with the data modeling literature and define an object as either an object-class or an object-instance. (Monarchi et al., 1988) An object-class is:

> . . . a named collection of properties and procedures. The value of a property may be atomic, a simple or composite object-instance, or a simple or composite object-class. A procedure may be associated with a specific property or with the object-class itself. (Monarchi et al., 1988, p. 3)

An object-instance is:

> . . . a member of an object-class. The description of the object-class defines the properties and procedures of individual members. Each member supplies values for the properties. (Monarchi et al., 1988, p. 3)

From the definitions presented above, the reader should realize that an object (either an object-class or an object-instance) is a conceptual entity that can be described both in terms of its static attributes and its behavior. One can view an object-class as a template and an object-instance as a specific occurrence of a template with values assigned to properties. (Monarchi et al., 1988)

We also note that some people working with the object paradigm view object-classes as prototypes rather than templates. Treating an object-class as a prototype allows for the potential cancelation of properties. We have purposely decided to prohibit the cancelation of properties. Consequently, we have adopted the template definition for an object-class.

For simplicity, we will refer to object-classes as classes and object-instances as instances throughout the remainder of this paper.

Object-Oriented Programming

Object-oriented programming (OOP) is a programming technique that employes the object paradigm. Classes and instances are directly represented in object languages. Static attributes associated with these entities are represented as data properties. Behavior is modeled by procedures. These procedures are commonly referred to as either class or instance methods. In object-oriented programming, methods are responsible for both sending and receiving messages.

OOP is useful because it enables the developer to build models that more closely resemble the real world. (Borgida et al., 1985) This natural correspondence between the real world and the computer model enable the programmer to build constructs in a semantically richer, more flexible programming environment than that offered through conventional programming techniques. Among the features of the object paradigm that provide the programmer with this flexibility are encapsulation, information hiding, polymorphism, data abstraction, class abstraction mechanisms, inheritance, and dynamic binding.

Encapsulation

According to Ledbetter and Cox (1985, p. 308), "Encapsulation defines a data structure and a group of procedures for accessing it." In other words, encapsulation is a concept that allows programmers to treat objects as self-contained, autonomous entities. It enables an object program to send messages to classes and instances without having to know the details behind an object's response to a message. Conversely, an object programmer can build classes and instances without having to worry about unwanted side effects affecting other objects. The only way a class or instance can interact with other objects is through the formal message passing protocol.

Information Hiding

A concept related to encapsulation is the idea of information hiding. Geoffrey Pascoe (1986, p. 308) describes this as: "Information hiding . . . [is a feature whe·e] the state of a software module is contained in private variables, visible only from within the scope of the module."

Given the definition above, information hiding is nothing more than the concept of using locally defined variables in a procedure. As far as the object paradigm is concerned, the scope of a local variable is extended to include the entire object. While the information hiding concept is certainly not new, it continues to be a powerful programming feature. Information hiding enables objects to have unlimited access to their own local state, while at the same time controlling the degree their local attributes are made available to other entities in a programming environment.

Polymorphism

Another feature related to the encapsulation concept is the idea of polymorphism. Stefik and Bobrow (1986, p. 41) describe polymorphism as:

> In general the term polymorphism means "having or assuming different forms," but in the context of object-oriented programming, it refers to the capability for different classes of objects to respond to exactly the same protocols [messages].

Stated more simply, polymorphism is a feature that allows different objects to respond differently to the exact same message. This feature enables object programmers to develop standardized interfaces between entities while at the same time having the freedom to implement an object's unique response to a message.

Data Abstraction

Data abstraction is a logical extension of the encapsulation concept. Stefik and Bobrow (1986, p. 41) describe data abstraction as, "The principle . . . that calling programs should not make assumptions about the implementation and internal representations of data types they use."

Data abstraction allows object programmers to implement primitive constructs and then use these lower level concepts to build increasingly powerful, more generalized data structures. In terms of object programming, these more generalized constructs often are represented as classes. The class relies on the implementation details its own methods and variables to exhibit the behavior of the abstracted data type.

Data abstraction enables the object programmer to concentrate on the level of detail appropriate for the given application. All the programmer need do is send messages to an object. He/she does not have to worry about the underlying implementation details.

Class Abstraction Mechanisms

Class abstraction deals with the relationships among classes; data abstraction is concerned with building higher level data constructs from lower level, implementation dependent primitives. Two main abstraction mechanisms prevalent in the object literature today are classification and generalization.

Classification. This abstraction mechanism is described by Gibbs (1985): "[Classification] allows one to ignore the details of particular objects by using a construct which represents a set of objects with similar structure."

This definition says that a class can be created based on the existence of similar instances. The attributes of the resulting class are shared among all its instances. In other words, classification connects instances to their defining class.

Generalization. Generalization combines similar properties of classes and stores them at a higher, more generic level. Alagic (1986) describes generalization as ". . . extracting common properties of one or more entity types [classes] while suppressing the differences among them."

Generalization allows the object programmer to build hierarchies of classes where a parent class (superclass) contains methods and variables common to all of its children (subclasses). Generalization is especially useful because it enables the programmer to store with a class information that is unique to the class. The programmer can then rely on class inheritance to spread the properties to the class's subclasses.

Inheritance

Inheritance allows classes to acquire properties of other classes based on their position in the class hierarchy. More specifically, subclasses inherit methods and variables from their superclasses. Inheritance frees the object programmer from having to duplicate specifications throughout the entire class hierarchy. Instead, all the programmer need do is store the property once at the appropriate level in the class hierarchy. After the property is stored with a class, it is then copied to all the class's subclasses.

Inheritance is also used to describe the mechanism used by instances to acquire properties from their defining class. An instance inherits both instance methods and instance variables from its defining class.

Dynamic Binding

Dynamic binding in an object-oriented system enables the programmer to write code without having to worry about the contingencies of handling different data types. For example, assume a programmer is writing a routine that prints the contents of a queue data structure. Also assume this queue contains a mixture of binary integers, floating point numbers, and character strings. Using conventional programming techniques, the programmer must test for each data type and perform the appropriate transformations to convert the data into printable form.

In the object paradigm, however, each data type is treated as an object and thus is responsible for its own data-dependent behavior. All the programmer needs to do is send a message to the appropriate data type to "print itself." The methods associated with the particular data type then perform the necessary transformations.

We refer to this concept as "dynamic binding" because new data types can be added to an object-oriented system without having to modify existing methods. The data-dependent code is automatically bound to existing methods because of the message passing protocol.

CHAPTER II

THE GALAHAD LANGUAGE SPECIFICATION: PRIMARY MODELING REQUIREMENTS

The modeling requirements for the Galahad Language were developed by professors Monarchi and Tegarden. Their requirements have evolved over the past year as they used the object paradigm to address conceptual modeling issues. Their original intent was to use existing object-oriented languages to pursue their research. Unfortunately, the capability associated with these object-oriented languages failed to provide the explicit support needed for addressing certain conceptual modeling concepts. Consequently, they devised the requirements for a language based on their own specific modeling needs.

The purpose of this chapter is to describe these modeling requirements. These requirements form the basis of the Galahad language. While it is beyond the scope of this paper to present the rationale behind the requirements, we will provide the reader with an explanation of the specific features Galahad supports. The WAR-SHIP examples we present with some of our explanations were inspired by Michael Hammer and Dennis McLeod's articles on the Semantic Database Model (SDM). (1978, 1981)

Galahad is based on the need for a conventional object-oriented language with explicit support for the following concepts: lattice inheritance, classification anomalies, instance anomalies, abstract superclasses, simple classes, domain/cardinality constraint enforcement, method specialization, the access-to mechanism, interclass existence dependencies, and instance aggregations. Each one of these requirements is described in more detail below.

Conventional Object-Oriented Concepts

Galahad supports the conventional object paradigm by directly modeling classes, instances, and message passing protocols. Galahad classes store their local state and exhibit their own behavior by using class variables and class methods. Similarly, Galahad instances use instance variables and instance methods to model their own status and performance as well. The reader is directed to the sections in Chapter 1 on the object paradigm and object-oriented programming for a brief overview of conventional object-oriented concepts.

Lattice Inheritance

Galahad programmers have the ability to arrange classes in a lattice network. Unlike a purely hierarchical approach, a lattice enables a class to have more than one immediate superclass. This way, a class with multiple parents can inherit properties directly from both parents. Examples of other languages that support lattice inheritance are FLAVORS (Moon, 1986) and SCOOPS (Texas Instruments, 1987b).

While multiple inheritance provides the programmer with a powerful tool for modeling classes, it does raise the issue of inheritance conflicts. Figure 2-1 illustrates this concept.

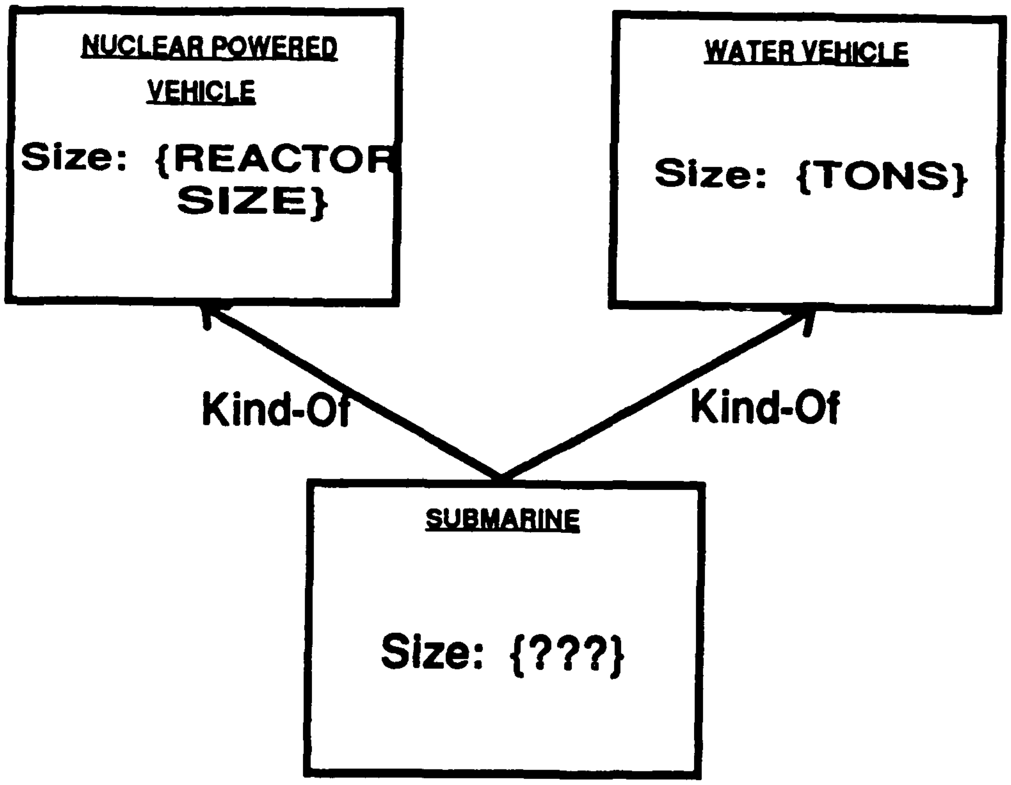


Figure 2-1. Lattice Inheritance Example

In this example, SUBMARINE is a subclass of both NUCLEAR POWERED VEHICLE and WATER VEHICLE. From which superclass should SUBMARINE inherit the attribute Size?

A common solution is to give precedence to the first listed (left most) superclass. In this case, SUBMARINE would inherit Size from NUCLEAR POWERED VEHICLE. The terminology for this form of conflict resolution is "depth-first up to joins" (Stefik and Bobrow, 1986, p. 48). The reason for this terminology is the system pursues a depth-first search, ignoring previously visited nodes. Galahad uses "depth-first up to joins" as a default for conflict resolution.

Galahad also provides explicit support for selectively overriding the "depth-first up to joins" conflict resolution mechanism. In the event the standard inheritance protocol is not adequate, a Galahad programmer can identify individual properties from specific superclasses to be included with the subclass. Referring again to figure 2-1, we can specify that Size in SUBMARINE can be inherited from WATER VEHICLE instead of from NUCLEAR POWERED VEHICLE.

Instance Lattice Inheritance

For conceptual modeling purposes, there occasionally exists the need to represent instances as "one of a kind" items. Grady Booch summarizes this need in the following statement: "In some cases, the class of an object may be anonymous . . . . The implication is that there may be only one object of the class (since there is no class name from which instances may be declared)." (1986, p. 216)

Instead of building a single class with only one instance, a Galahad programmer can represent these "one of a kind" instances directly. To better understand this concept, consider the example shown in figure 2-2.

In this example, we have one instance, DPT (short for Professor David P. Tegarden), and two classes, PROFESSOR and STUDENT. Notice that DPT is an instance of both PROFESSOR and STUDENT. There is no intermediate subclass between DPT, PROFESSOR, and STUDENT. By directly modeling instances in this fashion, the Galahad programmer is freed from the burden of building a class just to model an anomaly.

Instance Anomalies

Just as there is a need to model classification anomalies, there exists the requirement to represent individual instance anomalies as well. For our purposes, we define an instance anomaly to be an instance-object which requires methods and/or variables to be uniquely associated with the instance. These instance-specific methods and variables are directly attached to an instance. They have no corresponding specification in the instance's defining class.

To illustrate this concept, consider figure 2-3. In this example, we have the class CHESS-PLAYER with the attributes Name, and National-Standing. Associated with CHESS PLAYER is an instance with values Gary Kasparov and "1" assigned to the variables Name and National-Standing respectively.

Notice the instance has an additional property, Playing-Strategy. Playing-Strategy is a method that contains the unique chess strategies and tactics used by Gary Kasparov while playing a game. Since these strategies are unique to this single individual, there is no corresponding specification in the instance's

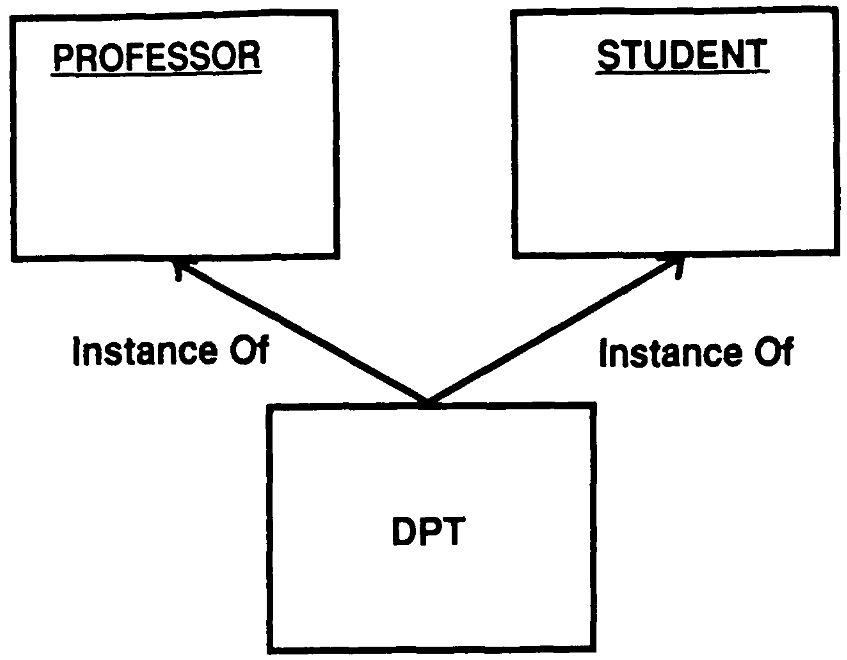


Figure 2-2. Instance Lattice Inheritance Example

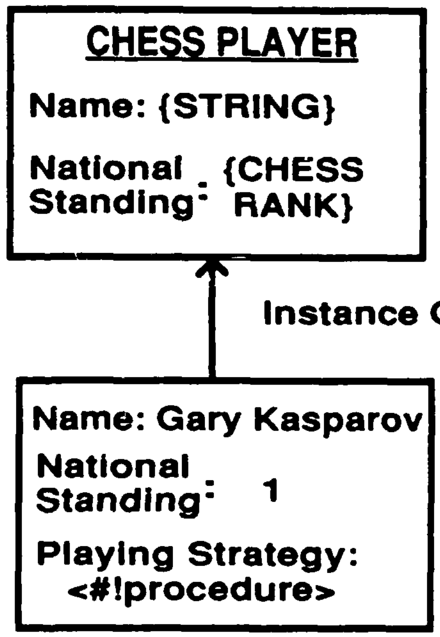


Figure 2-3. Instance Anomaly Example

defining class. Playing-Strategy is an instance-specific method. Additionally, any instance variables that are uniquely referenced by Playing-Strategy are considered to be instance-specific variables.

Galahad supports the modeling of instance anomalies. A Galahad programmer can assign instance-specific variables and methods to an instance. To the best of our knowledge, the only other object-oriented language providing explicit support for instance specific methods and instance specific variables is Object LOGO (Schmucker, 1986).

Abstract Superclasses

Galahad provides the programmer with the capability to model abstract superclasses. Adele Goldberg and David Robson give this definition of an abstract superclass:

> Abstract superclasses are created when two classes share a part of their descriptions and yet neither one is properly a subclass of the other. A mutual superclass is created for the two classes which contains their shared aspects. This type of superclass is called *abstract* because it was not created in order to have instances. (1983, p. 66).

As mentioned above, abstract superclasses enable the modeler to use generalization abstraction without worrying about the creation of unwanted instances. To illustrate, consider figure 2-4:

In this example, ENGINE is an abstract superclass of DIESEL ENGINE and GASOLINE ENGINE. (The dashed box around ENGINE signifies that the class is abstract.) Assuming we want only to model instances of DIESEL ENGINE and GASOLINE ENGINE, we are justified in creating ENGINE as an abstract superclass. This way, the integrity of the model is maintained by ensuring that instances of ENGINE cannot be built.

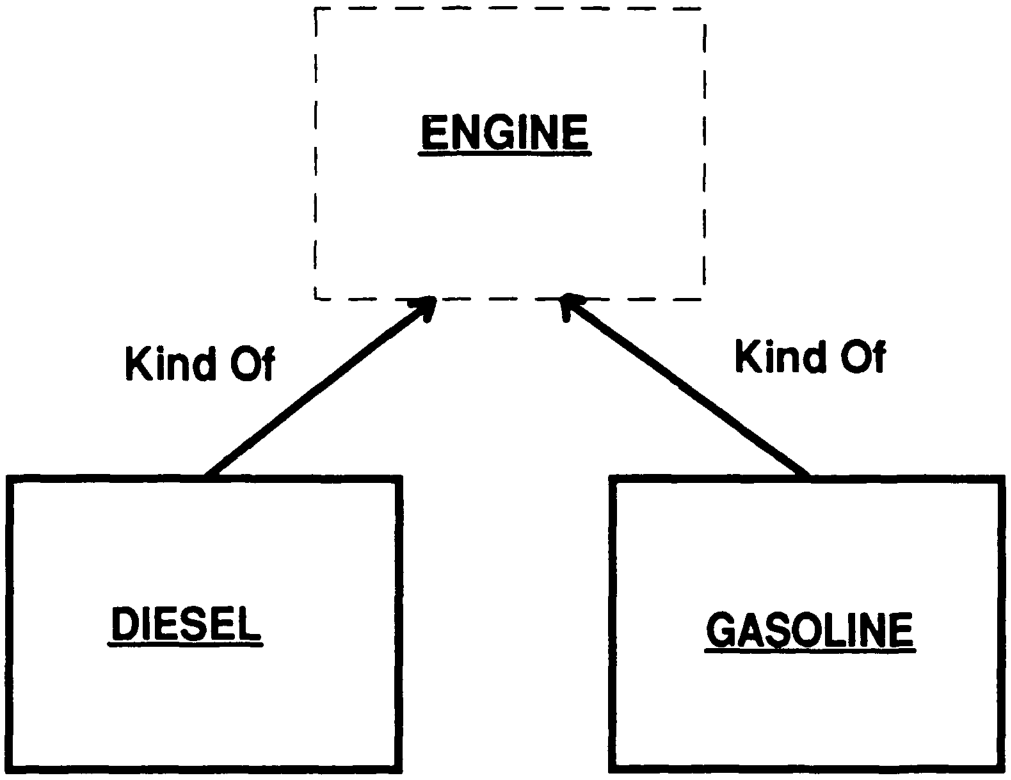


Figure 2-4. Abstract Superclass Example

Galahad supports the abstract superclass primitive by enabling the programmer to label a class as ABSTRACT. Galahad specifically prevents the creation of instances from ABSTRACT labeled classes.

Simple Classes

Galahad simple classes are used for the same purpose as that employed by name classes in SDM (Hammer and McLeod, 1981). Simple classes assist the Galahad programmer in building primitives designed to communicate with human users. Michael Hammer and Dennis McLeod describe name classes in terms of the Semantic Data Model as follows:

> In the real world, entities can be denoted in a number of ways; for example, a particular ship can be identified by giving its name or its hull number, by exhibiting a picture of it, or by pointing one's finger at the ship itself . . . However, there must also be some mechanism that allows for the outside world (i.e., users) to communicate with an SDM database. This will typically be accomplished by data being entered or displayed on a computer terminal. However, one cannot enter or display a real entity on such a terminal; it is necessary to employ representations of them for that purpose. (1981, p. 361)

In terms of object-oriented programming, a Galahad simple class is not a true object-class because it has no class/instance variables and no class/instance methods. Instead, it is a procedural specification designed to test the proper formatting of a passed object. This procedure is used to determine whether or not a passed object is a "member" of the simple class.

For example, the simple class NAME may be defined as a procedure that tests the length of a passed string and returns TRUE if the string's length is less than or equal to 15 characters. Assuming we wanted to test the string "David", "David" would pass the format test because it has less than 15 characters.

Therefore, Galahad would consider "David" to be a member of the simple class NAME.

Domain/Cardinality Constraint Enforcement

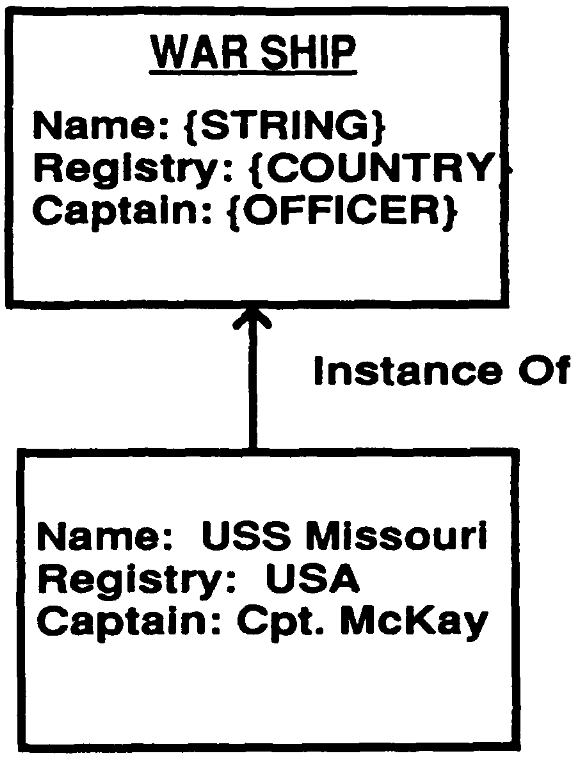
Galahad programmers have explicit support for enforcing domain and cardinality constraints on class and instance variables. This support enables a modeler to restrict the values a variable can assume based on specifications defined in the variable's defining class. To illustrate, refer to figure 2-5.

In this example, we have specified that instance variable Captain in class WAR-SHIP can assume values only from the class OFFICER. Additionally, we have designated all WAR-SHIPS are to have exactly one captain (cardinality (1 1)). As the reader can see, by restricting the values a class or instance attribute can assume, Galahad provides the modeler with an automatic means of ensuring variables do not violate domain or cardinality constraints.

A variable's legal domain and cardinality restrictions are defined during class creation. Galahad domains can be regular classes or simple classes. Cardinality is defined as a lower and upper bound pair. For example, (1 3) specifies a lower cardinality of 1 and an upper cardinality of 3.

Specialized Methods

Occasionally a Galahad programmer may need to build a class or instance method whose implementation depends on the class of the argument(s) passed to the method. For example, the class WAR-SHIP may have a method called Defend-Self. Defend-Self is a fairly complex method because the defense of a ship is dependent on the specific nature of the threat. While the modeler can



War Ship Cardinality

Name: (1 1)
Registry: (1 1)
Captain: (1 1)

Figure 2-5. Domain/Cardinality Constraint Example

certainly use a case statement to build one large method, it is generally more practical to write separate functions which respond specifically to one class of threat. Figure 2-6 illustrates this concept.

In this example, the actions a WAR-SHIP takes to defend itself depend specifically on the type of threat to the vessel. In terms of object programming, the Defend-Self method is really four separate procedures. Three of the four procedures model behavior based on the specific nature of the threat. The fourth is a dispatching function that checks on the class of the threat (in this case missile, ship, or aircraft), and invokes the proper implementation-specific procedure that "defends" the ship.

Galahad provides explicit support for the specialization of methods based on the classes of all message arguments sent to an object. The original idea came from the Common Lisp Object System (CLOS). (Keene, 1989) CLOS uses a similar dispatching function that uses the class of a message's arguments to call the appropriate procedure.

Galahad extends this concept by allowing the programmer to specialize optional message arguments as well. Optional arguments are specialized only if they are present. For example, assume Defend-Self has an optional parameter which indicates the priority to be assigned to the threat. If no priority is specified to the dispatching function, the system ignores the optional parameter, assumes HIGH urgency, and dispatches the appropriate method. Conversely, if a LOW or MEDIUM priority was assigned and passed as an optional parameter, Galahad specializes on the parameter and invokes the appropriate method.

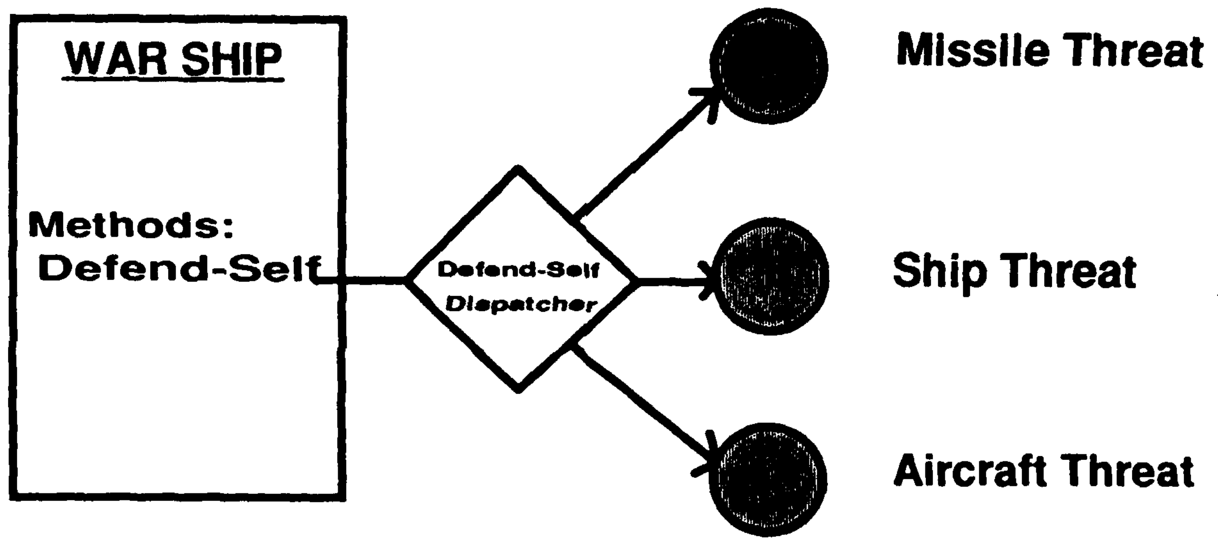


Figure 2-6. Specialized Methods Example

The Access-To Mechanism

A concept closely related to specializing methods is the issue of controlling access to an object. Grady Booch describes this access-to mechanism in terms of restricting the scope of objects.

> The names of objects should have a restricted scope. Thus, in designing a system, we concern ourselves with what objects see and are seen by another object or class of objects. . . . In the worse case, all objects can see one another, and so there is the potential of unrestricted action. It is better that we restrict the visibility among objects, so as to limit the number of objects we must deal with to understand any part of the system and also to limit the scope of change. (1986, p. 216)

In addition to limiting scope, the access-to mechanism includes the idea that individual objects behave differently based on the class of the object sending the message. For example, in our simulation of the WAR-SHIP environment, the method Defend-Self should only be invoked by messages from "authorized" objects. In other words, we don't want any arbitrary object to send a message to a WAR-SHIP to defend itself.

Galahad explicitly supports the access-to concept. A Galahad programmer can specialize a method to respond only to a certain sender or class of senders. By specializing methods according to the sender of the message, Galahad helps ensure continued model integrity.

Interclass Existence Dependencies

Galahad programmers have explicit support for modeling existence dependencies among classes. To illustrate this concept, Peter Chen describes an existence dependency in terms of a relationship between an EMPLOYEE and an employee's DEPENDENTs.

> [We] . . . can express the existence dependency of one entity type on another. For example, the . . . relationship set EMPLOYEE-DEPENDENT indicates that existence of an entity in the entity set DEPENDENT depends on the corresponding entity in the entity set EMPLOYEE. That is, if an employee leaves the company, his dependents may no longer be of interest. (1976, p. 20)

To illustrate further, refer again to figure 2-5. Assume that the modeler is interested only in representing the class WAR-SHIP. In this case, OFFICER would be existent dependent on WAR-SHIP because if the class WAR-SHIP were removed from the system, there would be no need to continue representing CAPTAIN.

Galahad supports dependent classes by allowing the modeler to specify at class creation time whether a class is existent dependent on another class. If a class has been defined to be dependent on another, then the dependent class is automatically removed from the system whenever the corresponding non-dependent entity is deleted.

Galahad supports dependent relationships at the instance level as well. For example, if an instance has been defined to be dependent on another, then the dependent instance is automatically removed from the system whenever the corresponding non-dependent instance is deleted.

Instance Aggregation

The Galahad programmer may need an abstraction mechanism that models aggregates, or collections, of objects. Additionally, the modeler may require a means of relating instances of one particular class to instances of another class. For example, consider figure 2-7.

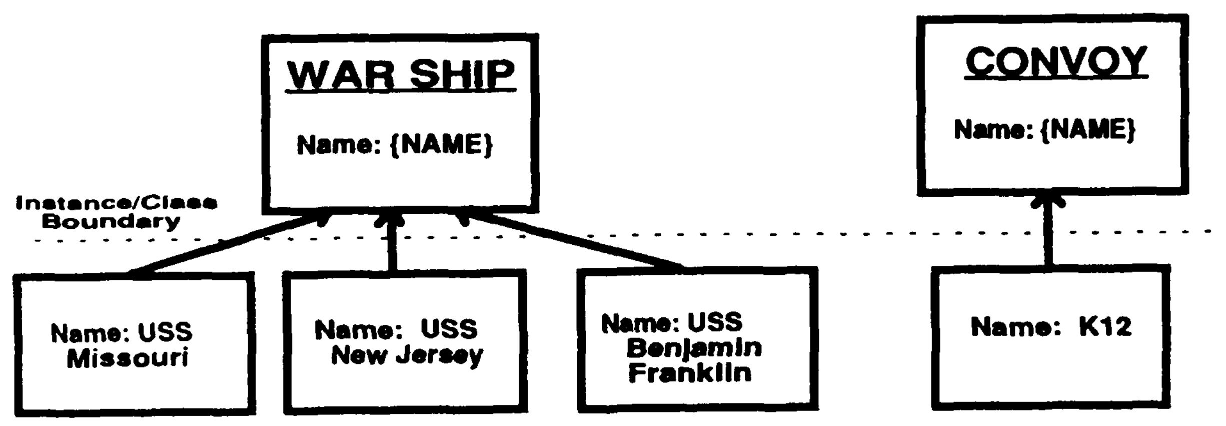


Figure 2-7. Instance Aggregation Example

In this example, we have the class WAR-SHIP and three instances: USS Missouri, USS New Jersey, and the USS Benjamin Franklin. We also have the class CONVOY and an instance of convoy, K12. Assume the three instances of WAR-SHIP become members of convoy K12. Galahad provides explicit support for showing the relationship between the three ship instances and the instance of convoy. This relationship is referred to as an instance aggregation. (Monarchi et al., 1988)

In the text below, Monarchi et al. describe the term instance aggregation and relate it to other abstraction mechanisms:

> Part, Element, Member. We refer to these three mechanisms as instance aggregations because they relate instances of classes to other instances. (Generalization relates classes to classes; classification relates instances of a class to their defining class; and property aggregation groups properties into a class). (1988, p. 8)

Notice their definition partitions instance aggregation into three types: part, element, and member. While the overall concept of these instance aggregations is the same, each has its own unique characteristics.

Part Aggregation

Part Instance Aggregation is used as a means to describe physical, structural relationships. A more concise definition is:

> Part aggregation groups instances of classes together to form an instance of a new class. In contrast to generalization, the new class is not a superclass of its component classes. This type of aggregation is represented by the Part-of relationship between components and the aggregate. Its inverse is Parts. (Monarchi et al., 1988, p. 9)

An example of a part aggregation would be the relationship between a specific automobile and its comprising structural components such as doors, hoods, etc.

Also, from the definition above, the reader should notice that an aggregate instance cannot exist without the prior existence of its component parts. Using the example above, this is analogous to saying an automobile cannot exist if none of its comprising parts exits.

Element Aggregation

Element Instance Aggregation is used to describe logical, nonstructural relationships. Specifically,

> Element aggregation also groups instances together to form an instance of a new class. And like part aggregation, the new class is not a superclass of its components' classes. Bus in contrast to part aggregation, element aggregation groups instances together when the association is logical rather than structural. This type of aggregation is represented by the Element-of relationship between the components and the aggregate; its inverse is Elements. (Monarchi et al., 1988, p. 10)

An example of an element aggregation would be the assignment of a military officer to a ship. Assuming an assignment must have both an officer and a ship to exist, a specific officer and a specific ship are considered to be Elements of the assignment.

Member Aggregation

Member Instance Aggregation is a specialization of the Element Aggregation concept.

> Member aggregation groups multiple instances of a class into an instance of a new class. It is [a] special case of the Element-of relationship because all instances which are grouped together to form a new aggregate are from the same class. And just as in Element-of, the grouping is logical rather than structural. . . Member aggregation is represented by the Member-of relationship between components and the aggregate. The inverse is Members. (Monarchi et al., 1988, p. 11)

The convoy example presented in figure 2-7 can be modeled using Member aggregation. In this particular case, the three instances of WAR-SHIP become Members of CONVOY K12.

Concluding Remarks on Instance Aggregation

Galahad provides explicit support for Part, Element, and Member aggregations. Galahad programmers specify at the class level the permitted instance aggregation connections between classes. For example, the modeler will specify that a CONVOY has as its Members instances of WAR-SHIP. Galahad then enforces these specifications during the creation and modification of instances. This way, the programmer is prevented from making a undesired instance aggregation connection. Also, since instance aggregations can have cardinality specifications, Galahad enforces cardinality constraints as well.

CHAPTER III

SURVEY OF CURRENTLY AVAILABLE OBJECT-ORIENTED LANGUAGES

The modeling requirements presented in Chapter 2 need to be examined in the light of other currently available object-oriented languages (OOLs). The purpose of this chapter is to provide the reader with a brief overview of some of these languages. We also present a comparison table which enables the reader to contrast the Galahad specification with these other systems.

The languages we describe below were chosen for either their historical significance, or their influence on the formulation of Galahad's design. The specific languages we discuss are SIMULA, Smalltalk, Actor, Flavors, Loops, SCOOPS, and the Common Lisp Object System (CLOS). At the end of the language descriptions, we present the table which compares the major features of each of these languages.

SIMULA

SIMULA (SIMUlation LAnguage) was developed in the middle 1960's by O.J. Dahl and K. Nygaard of the Norwegian Computing Center, Oslo, Norway. The language was designed explicitly to support discrete event simulation. It also was the first language to employ object-oriented concepts.

SIMULA, which is based on ALGOL, enables the programmer to model systems by allowing for the creation of classes and instances. Also, the programmer is free to arrange classes in a generalization hierarchy. This enables subclasses to inherit procedures and variables from their appropriate superclass.

Since SIMULA is a simulation language, all activity in a SIMULA program is modeled by a series of discrete transactions. A transaction in SIMULA is an entity that moves through the simulation model and interacts with the model's various components. For example, a SIMULA program that models a traffic intersection would have automobiles and pedestrians as individual transactions in the "moving" portion of the model.

In light of the discussion above, it is important to realize that a transaction in SIMULA is different from a message in today's object-oriented systems. A SIMULA transaction is an instance of a class. It is used as a "unit of traffic" to model the overall behavior of a system. A message is merely a means of formalized communication among objects. Unlike SIMULA transactions, messages are not objects.

Since transactions are crucial to a successful simulation, SIMULA classes are specifically designed to facilitate the movement of transactions in the model. Consequently, they don't have individual methods which can respond differently to different messages. Instead, a class specification is defined in one BEGIN/END block and all the code is dedicated to the movement of transactions within the model.

Another unique feature of SIMULA is the explicit support for the modeling of time. Since SIMULA models discrete events, the language must have the primitives necessary to monitor the passage of time and update the state of the system as necessary.

Smalltalk

Smalltalk was originally designed in the early 1970s by the researchers at Xerox PARC in Palo Alto California. The vision for the language came from Alan Kay who was searching for an easy, fun to use system to accompany his Dynabook project. Since the early 70s, Smalltalk has gone through several iterations, the latest of which is embodied in Smalltalk-84.

Smalltalk takes the credit for being the first language completely designed and implemented using the object paradigm. Unlike SIMULA, which uses ALGOL statements to model data, all of Smalltalk's data primitives are implemented as classes. By representing everything as an object, Smalltalk provides a uniform environment for building programs. Additionally, Smalltalk has an extensive programming library which saves the programmer from having to write any low-level input/output routines.

Smalltalk provides explicit support for the creation of classes, abstract classes, instances, and methods. The language enables the programmer to arrange classes in a hierarchy to allow for the inheritance of methods and variables. Smalltalk-80 (Version II) has extended the inheritance concept to include lattice inheritance as well.

Smalltalk also allows the programmer to model class, pool, and global variables. Class variables are variables which apply to a particular Smalltalk class. A class variable is also shared by all instances of a class. Pool variables are similar to class variables; however, they are shared by all instances of a predefined group of classes. Finally, global variables are data shared by all instances in a Smalltalk application.

ACTOR

Actor is an object-oriented language originally developed in 1986 by the Whitewater Group in Evanston, Illinois. Actor operates as an application under MS-Windows and is one of the first OOLs to bring an integrated object-oriented programming environment to personal microcomputers.

Actor provides modeling support for classes, instances, instance variables, instance methods, class methods, and standard hierarchical inheritance. Conspicuously absent, however, is the support for class variables. If the programmer desires to use a variable that either pertains to a class, or is shared by more than one instance, he/she is forced to use a global variable. Global variables are data which are shared by all objects in an Actor application.

Like Smalltalk, Actor's design is completely embedded in the object paradigm. All data objects are implemented as Actor classes. Also, Actor parallels Smalltalk by having a master superclass called **Object** from which all globally available methods are inherited. Actor implements Smalltalk's **Metaclass** concept by using a separate **Behavior** class. **Behavior** contains all the class methods necessary for the creation and management of user-defined Actor classes.

While Actor is considered a "pure" object system, it does make a fundamental departure from the conventional object paradigm. Specifically, any object in Actor can access another object's instance variables without using the message passing protocol. This direct accessing of another object's variables violates the fundamental principles of data hiding and encapsulation. The Whitewater Group acknowledges this weakness by stating:

> This practice [of direct access] is discouraged in cases where altering an object's internal state could produce complicated side-effects. It is an easy and efficient way of communication with an object, but should be used with caution. Many benefits accrue from letting complex objects manage their own state. (1987, p. 543)

Hopefully, the next version of the Actor language will correct this fundamental design flaw.

FLAVORS

Flavors is an OOL which was originally developed by the MIT Lisp Machine group in 1979. It went through major revisions in both 1981 and 1985, and now it is used in nearly every aspect of the Symbolics 3600 Operating System. Like Smalltalk and Actor, Flavors has its own fully integrated programming environment, complete with editors, browsers, compilers, and debuggers.

Flavors allows the programmer to model classes, instances, instance variables, instance methods, and lattice inheritance. An especially interesting feature is that Flavors sticks with the sensory paradigm when defining classes. Classes are descendents of **Vanilla**, a class similar in function to Smalltalk's **Object** class. Also, since Flavors allows for lattice inheritance, subclasses are described as being "mixed" from several component (superclass) flavors. The resulting class is now viewed as a new flavor, based in part on the mixture of the component superclasses.

A class in Flavors arranges its component superclasses in most specific to most general order. By arranging components in this manner, the system has a framework for basing the resolution of inheritance conflicts. A default for conflict resolution is to give priority to the most specific superclasses; however, a Flavors programmer can change this strategy to suit his/her individual needs.

This flexibility is especially apparent when working with a class's methods. The Flavors programmer has a wide degree of latitude in specifying the exact combination of methods to be called in response to any particular message. This combination of methods can include permitting a **before-method** to be called prior to the **primary method**. The programmer can also specify the invocation of an **after-method** following termination of the **primary method**. To provide even further control, Flavors allows the programmer to specify conflict resolution rules should there be conflicting **before, primary**, or **after-methods** in a class's superclasses.

Another interesting feature of the Flavors language is the dynamic nature of Flavors classes. A modeler can alter a particular class in Flavors, and the Flavors system will automatically update the structure of the class's instances.

Loops

Loops was developed in 1983 by Daniel Bobrow and Mark Stefik at Xerox PARC in Palo Alto California. The language design was based on the notion that no single programming model provides elegant solutions to all types of programming pr blems. Consequently, Loops was developed with elements of the procedural, object, data access, and rule-based programming paradigms. The intent was to provide the programmer with a single language incorporating aspects of all four of these models. This way, Loops could (possibly) be used to solve all classes of programming problems.

In terms of object-oriented programming, Loops provides the programmer with the means to model classes, abstract classes, simple classes, and instances. For the creation of classes, the modeler can use class variables, instance variables, and instance methods. The programmer can also arrange Loops classes in a lattice network. Loops uses the conventional depth-first left to right prioritizing mechanism for the resolution of inheritance conflicts.

In addition to the above capabilities, Loops provides the programmer with several features that address specialized modeling needs. These features include the assignment of property lists to variables, the use of active values, the modeling of composite objects, and specific support for viewing an object from several different "perspectives."

Of the four features mentioned above, the two most interesting are the modeling of composite objects, and viewing classes and instances from different "perspectives." Composite objects enable the programmer to establish an existence dependency between two or more instances. This is the same existence dependency we discussed in Chapter 2. Whenever a Loops composite object is instantiated, it automatically causes its component objects to be instantiated. The "perspective" view mechanism also is a very powerful feature because it enables the modeler to work with an object and keep separate the information which is used for different purposes. Loops can differentiate among different perspectives based on the class's immediate parents in the inheritance lattice.

SCOOPS

SCOOPS (Scheme Object-Oriented Programming System) is an object-oriented extension to TI Scheme, a version of the Scheme programming language developed by Texas Instruments. SCOOPS, developed in 1985, is similar in both form and function to Loops and Flavors. The language reflects the object paradigm by providing support for modeling classes, instances, and methods.

Probably the two most powerful features of the language are SCOOPS' abilities to model lattice inheritance and active values. SCOOPS supports lattice inheritance at the class level. SCOOPS classes inherit variables and methods from their superclasses by pursuing a depth-first search up the lattice network. Priority is given to those classes appearing first in a class's superclass list. Name conflicts are resolved by a SCOOPS class adopting the first occurrence in the inheritance network.

Another powerful feature of SCOOPS is the language's ability to attach active values to instance variables. Active values provide the modeler with the added flexibility of automatically calling a procedure whenever an instance variable is referenced. This added control feature increases the capability of the language by providing programmers with a means of tieing an object's behavior directly to the accessing of its variables.

One major disadvantage of the language is that SCOOPS does not allow the programmer to treat a class as a true object. No capability exists for the SCOOPS programmer to create user-defined class methods. Since there is no provision for class methods, no messages can be sent directly to a SCOOPS class. Instead, to create instances and access class variables, the modeler is forced to break from the message passing paradigm and use the globally available procedures MAKE-INSTANCE, SET-CV, and GET-CV. The use of these procedures leads to an inconsistent user interface.

CLOS

CLOS (Common Lisp Object System) was developed in direct response to the need for a standardized object-oriented language. In 1986, a CLOS working group was formed to devise a standard object language for adoption by the ACM X3J13 Committee on the formal standardization of Common Lisp. This group, which included Daniel Bobrow, Richard Gabriel, and David Moon, devised a language based on the most useful and well established features of other currently available object-oriented languages. Since that time, the X3J13 committee has adopted the CLOS specification as a standardized object-oriented extension to the Common Lisp language.

Even though CLOS is a totally new language, it still has most of the constructs associated with conventional object-oriented programming. Specifically, a programmer can model classes, instances, class/instance variables, and class/instance methods. A CLOS programmer can also allow for multiple inheritance by arranging classes in a lattice network. CLOS follows Flavors's lead by resolving inheritance conflicts based on a class precedence list. In CLOS, like Flavors, a class's precedence list is arranged in most specific to least specific order.

A major departure from the conventional object paradigm, however, is that CLOS no longer "sends" messages to destination objects. Instead, the language has borrowed from Flavors and adopted a "generic function" idea. A CLOS generic function is called within a method just like any other conventional Lisp function. Unlike a standard Lisp function, however, a CLOS generic function is responsible for "dispatching" the proper method based on the parameters passed to the generic function. This concept is identical to our Specialized Methods concept presented in Chapter 2. The only difference is that CLOS uses the generic function idea to completely replace formalized "sends". Galahad continues using the SEND concept to request service from an object; however, as soon as an object has received a message, it uses a generic dispatch function to call the appropriate specialized method.

Another unique feature associated with CLOS is the introduction of **around-methods**. An **around-method** enables the programmer to specify the exact ordering of a primary method's **before** and **after-methods**. Additionally, **around-methods** enable the programmer to override the standard inheritance mechanism and specify individual methods to be used from the inheritance lattice.

Comparison of Major Features of Current Object-Oriented Languages

Table 3-1 shows a matrix comparing the modeling capability of each of the above described languages and Galahad. The reader should note that a language is credited with possessing a particular feature only if it provides explicit support for the concept. We understand that many of the languages can be extended to provide the indicated modeling capability; however, except where noted, we restrict our comparison to explicitly supported features only.

To assist the reader in comparing each of the listed languages, we include a glossary briefly describing each feature listed in Table 3-1. This glossary is presented in Table 3-2. Except where noted, all items are discussed in more detail in the main body of this paper.

Table 3-1. Comparison of Object-Oriented Languages

	SIMULA	Smalltalk	Actor	Flavors	Loops	SCOOPS	CLOS	Galahad
Classes	YES	YES	YES	YES	YES	YES	YES	YES
Abstract Classes	NO	YES	NO	YES	YES	NO	YES	YES
Simple Classes	NO	YES	YES	NO	YES	NO	NO	YES
Instances	YES	YES	YES	YES	YES	YES	YES	YES
Message Passing	NO	YES	YES	NO^1	YES	YES	NO^1	YES
Class Hierarchical Inheritance	YES	YES	YES	YES	YES	YES	YES	YES
Instance Hierarchical Inheritance	YES	YES	YES	YES	YES	YES	YES	YES
Class Lattice Inheritance	NO	YES^2	NO	YES	YES	YES	YES	YES
Instance Lattice Inheritance	NO	NO	NO	NO	NO	NO	NO	YES
Instance Aggregations	NO	NO	NO	NO	NO	NO	NO	YES
Interclass Existence Dependencies	NO	NO	NO	NO	YES	NO	NO	YES
Class Variables	NO	YES	NO	NO	YES	YES	YES	YES
Class Methods	NO	YES	YES	NO	NO	NO	YES	YES
Class Specialized Methods	NO	NO	NO	NO	NO	NO	YES	YES
Instance Variables	YES	YES	YES	YES	YES	YES	YES	YES
Instance Methods	YES	YES	YES	YES	YES	YES	YES	YES

Table 3-1 (continued).

	SIMULA	Smalltalk	Actor	Flavors	Loops	SCOOPS	CLOS	Galahad
Instance Specialized Methods	NO	NO	NO	NO	NO	NO	YES	YES
Instance Specific Variables	NO	NO	NO	NO	NO	NO	NO	YES
Instance Specific Methods	NO	NO	NO	NO	NO	NO	NO	YES
Active Values	NO	NO	NO	NO	YES	YES	NO	YES[3]
Access-To Mechanism	NO	NO	NO	NO	NO	NO	NO	YES
Domain Constraints	NO	NO	NO	NO	NO	NO	NO	YES
Cardinality Constraints	NO	NO	NO	NO	NO	NO	NO	YES
User-Defined Class Constraints	NO	NO	NO	NO	NO	NO	NO	YES
User-Defined Instance Constraints	NO	NO	NO	NO	NO	NO	NO	YES
User-Defined Instance Specific Constraints	NO	NO	NO	NO	NO	NO	NO	YES

[1]Flavors and CLOS don't implement message passin , as formal "sends". Instead, they use generic functions which are responsible for dispatching individual methods.

[2]The original versic of Smalltalk-80 did not implement class lattice inheritance; however, later versions do.

[3]Galahad does not allow active values to be specified during instance/class variable creation; however it does support the concept by allowing the programmer to write his/her own GET and SET methods. References to active values can be included in these user-defined methods.

Table 3-2. Glossary of Terms

Class	An object-class is a grouping of properties which describe similar objects. Depending on one's perspective, it can be viewed as either a template or a prototype.
Abstract Class	An abstract class is a class that does not allow for the creation of instances. Instead, it is used to define properties that will be inherited in the generalization hierarchy.
Simple Class	A simple class is a printable data object. Simple classes are identical to the name class concept of the Semantic Data Model.
Instance	An instance is a member of an object-class. An instance has its own identity and supplies values to a class's properties
Message Passing	Message passing is the communication protocol of the object paradigm. Messages request behavior from objects. Messages can also be used to transmit information between objects.
Class Hierarchical Inheritance	Class hierarchical inheritance allows classes to acquire properties of other classes based on their position in a generalization hierarchy. In a class hierarchy, each class has only one superclass.
Instance Hierarchical Inheritance	Instance hierarchical inheritance allows instances to acquire properties from their defining class. In an instance hierarchy, instances have only one defining class.

Table 3-2 (continued).

Class Lattice Inheritance	Class lattice inheritance allows classes to acquire properties of other classes based on their position in a generalization network. In a class lattice, each class can have one or more superclasses.
Instance Lattice Inheritance	Instance lattice inheritance allows instances to acquire properties of their defining class(es). In an instance lattice, instances can have more than one defining class.
Instance Aggregations	An instance aggregation is a grouping of instances into an aggregate, or collection. It also is a means of relating instances to other instances.
Interclass Existence Dependencies	An interclass existence dependency is a relationship between two classes where one class cannot exist without the a priori existence of another class. This term also applies to an instance of one class depending on the prior existence of another instance.
Class Variables	A class variable is a property that pertains to the class rather than to instances of the class. It also is used to describe properties whose values are shared by all instances of a class.
Class Methods	A class method is a procedural specification that defines a class's behavior rather than the behavior of the class's instances.
Class Specialized Methods	A class specialized method is a class method that exhibits behavior only if the arguments passed in the message are of a pre-specified data type.
Instance Variables	An instance variable is a property that pertains to individual instances of the class.

Table 3-2 (continued).

Instance Methods	An instance method is a procedural specification that defines the behavior of a class's instances.
Instance Specialized Methods	An instance specialized method is an instance method that exhibits behavior only if the arguments passed in the message are of a pre-specified data type.
Instance Specific Variables	An instance specific variable is a property that pertains only to a single instance.
Instance Specific Methods	An instance specific method is a procedural specification that pertains only to a single instance.
Active Values	An active value is a procedural specification that defines an object's behavior each time a class or instance variable is accessed.
Access-To Mechanism	The access-to mechanism is a constraint that enables an object to respond differently to a message based on the identity of the sender. This differentiation of behavior includes "ignoring messages" as a possible response.
Domain Constraints	A domain constraint is a specification that restricts the legal values a class or instance property can assume. This restriction is based on the class membership of the value.
Cardinality Constraints	A cardinality constraint is a specification that restricts the values that can be assigned to a class or instance variable. This restriction is based on the permitted number of different values a property can assume at a given time.

Table 3-2 (continued).

User-Defined Class Constraints	User-defined class constraints are not covered in the text of the paper. These constraints are class-level restrictions that are dependent on the user's application. User-defined class constraints are procedural specifications. The Galahad programmer defines them in a manner similar to that used to create class methods.
User-Defined Instance Constraints	User-defined instance constraints are not covered in the text of the paper. These constraints are instance-level restrictions that are dependent on the user's application. User-defined instance constraints are procedural specifications. The Galahad programmer defines them in a manner similar to that used to create instance methods.
User-Defined Instance Specific Constraints	User-defined instance specific constraints are not covered in the text of this paper. These constraints are user-defined restrictions that apply to a single instance. Since user-defined instance specific constraints are procedural specifications, the Galahad programmer defines them in a manner similar to that used to create instance specific methods.

CHAPTER IV

THE GALAHAD SYSTEM

The Galahad system is designed specifically to provide Professors Monarchi and Tegarden a tool for conceptual modeling research. This chapter describes in technical terms how the system implements the modeling requirements given in Chapter 2. We begin by first providing the reader with a brief background of our development methodology and coding conventions. We then divide the remainder of the chapter into two major sections: Galahad Basics and Modeling Requirements Implementation.

The first section, Galahad Basics, introduces the reader to the data structures we use to represent classes and instances. We also present the structure of the Galahad system from both a logical and a programming perspective. We conclude this section by discussing the coding mechanisms behind sending a typical message in Galahad.

The second section, Modeling Requirements Implementation, discusses the implementation of the modeling specifications presented in Chapter 2. Where appropriate, we present the reader with segments of Galahad code. Since our discussion will be at the coding level, we assume the reader is familiar with the programming language TI Scheme.

Development Methodology

Galahad was developed using a prototyping design methodology. The conditions lent themselves well to this approach because of the small number of users, the relatively small size of the system, and the continually evolving requirements specification. Also, everyone in our group was committed to see the project through to completion.

For the purposes of system testing and acceptance, Professor Tegarden was designated as the primary user. His main task was to verify the functionality of the system and to make suggestions for improving the usefulness of the language. Six different versions of the Galahad system were delivered to Professor Tegarden for his evaluation. The average time between version deliveries was two weeks.

Closely associated with the choice of a prototyping development approach was our selection of an implementation environment. Successful prototyping requires rapid turnaround of successive versions of the system. (Pressman, 1987) In light of this fact, we chose PC Scheme, a specific implementation of TI Scheme from Texas Instruments. The hardware on which we operated was an IBM PS/2 model 50 (1 Hz. wait state).

Scheme is an extremely expressive language that has certain features allowing for the rapid development of software. For example, Scheme enables the programmer to treat all constructs in the language as first-class objects. This minimizes restrictions on program design. Also, all data types within the language are loosely-typed. This feature frees the programmer from formally declaring data before they are used by the program. Finally, Scheme supports an incrementally extended environment frame hierarchy. Scheme environments allow the programmer to limit the scope of Scheme program variables. Also, environment frames allow programmers to build incrementally-defined variable inheritance hierarchies. The incremental definition of an inheritance hierarchy is especially useful for those applications, such as Galahad, modeling generalization abstraction.

Finally, another reason for our choice of PC Scheme was our familiarity with the internal workings of the object-oriented extension to TI Scheme, SCOOPS. We spent considerable time examining the internal representation of SCOOPS classes, instances, and methods. This investigation provided us with a good frame of reference for beginning the implementation of Galahad. Appendix C presents the results of our SCOOPS analysis.

Coding Conventions

Since prototyping was our choice for a development methodology, the coding effort was oriented toward rapid turnaround of Galahad versions. Additionally, we wanted to create maintainable code for future enhancements. To achieve both of these goals, our code followed the object paradigm whenever possible. We specifically pursued the object paradigm during our creation of the OBJECT class.

Additionally, we purposely traded code compactness for code simplicity. For example during our implementation of the instance aggregation concept, we wrote separate functions to implement Part, Element, and Member aggregations. This functionality could have been collapsed into a single, more complex procedure; however, we felt it to be more understandable (and maintainable) to write separate, parallel code.

To improve code readability, we made extensive use of the LET statement to assign temporary variables. We also designed variable names to be as descriptive as possible. These two coding conventions tended to create rather lengthy functions; however, we felt maintainability was enhanced by pursuing a self-documenting programming style.

Galahad Basics

To best understand how we implemented the modeling requirements presented in Chapter 2, the reader must first have basic knowledge for the overall structure of the system. The purpose of this section is to provide this basic knowledge. We begin by introducing the data structures used to implement both Galahad classes and instances. We then describe the Galahad system from both a logical and a programming perspective. We conclude by discussing the mechanism behind sending a typical message in Galahad.

Galahad Primary Data Structures

Galahad classes and instances were implemented using Scheme's DEFINE-STRUCTURE statement. We chose to use scheme structures over environments because of our requirement to model instance lattice inheritance. To implement an instance lattice using environments would have required an environment frame to be connected to more than one parent. Lexical scoping in Scheme prohibits this.

We chose to use structures over standard Scheme vectors strictly for convenience purposes. The DEFINE-STRUCTURE statement automatically creates vectors, complete with default values and primitives to access each individual slot. For simplicity, we will refer to the data structures used to implement classes and instances as vectors throughout the remainder of this paper.

Classes. The primary data structure used to implement a Galahad class is a 30 slot vector. Figure 4-1 shows a class vector with each individual slot labeled. Appendix A, A Description of Galahad's Class and Instance Vectors, contains more detailed information about each component of a Galahad class vector.

The reader may note that the number of slots associated with a Galahad class vector is quite large. We purposely chose to create a separate slot for Part, Element, and Member aggregations. This was done both to simplify the code and to underscore our implementation of the instance aggregation concepts. We also chose to separate class and instance methods from class and instance constraints. Again, this was done for both maintenance and accent purposes.

Instances. The data structure used to implement a Galahad instance is a nine slot vector. Figure 4-1 shows an instance vector with each individual slot labeled. As with the class vector, we purposely chose to create a separate slot for Part, Element, and Member aggregations. The reader is referred to Appendix A for details on the individual components of an instance vector.

Galahad Logical Design

The Galahad system is divided into four functional components: SEND, CONTROL, METACLASS, and OBJECT. SEND and CONTROL serve more as

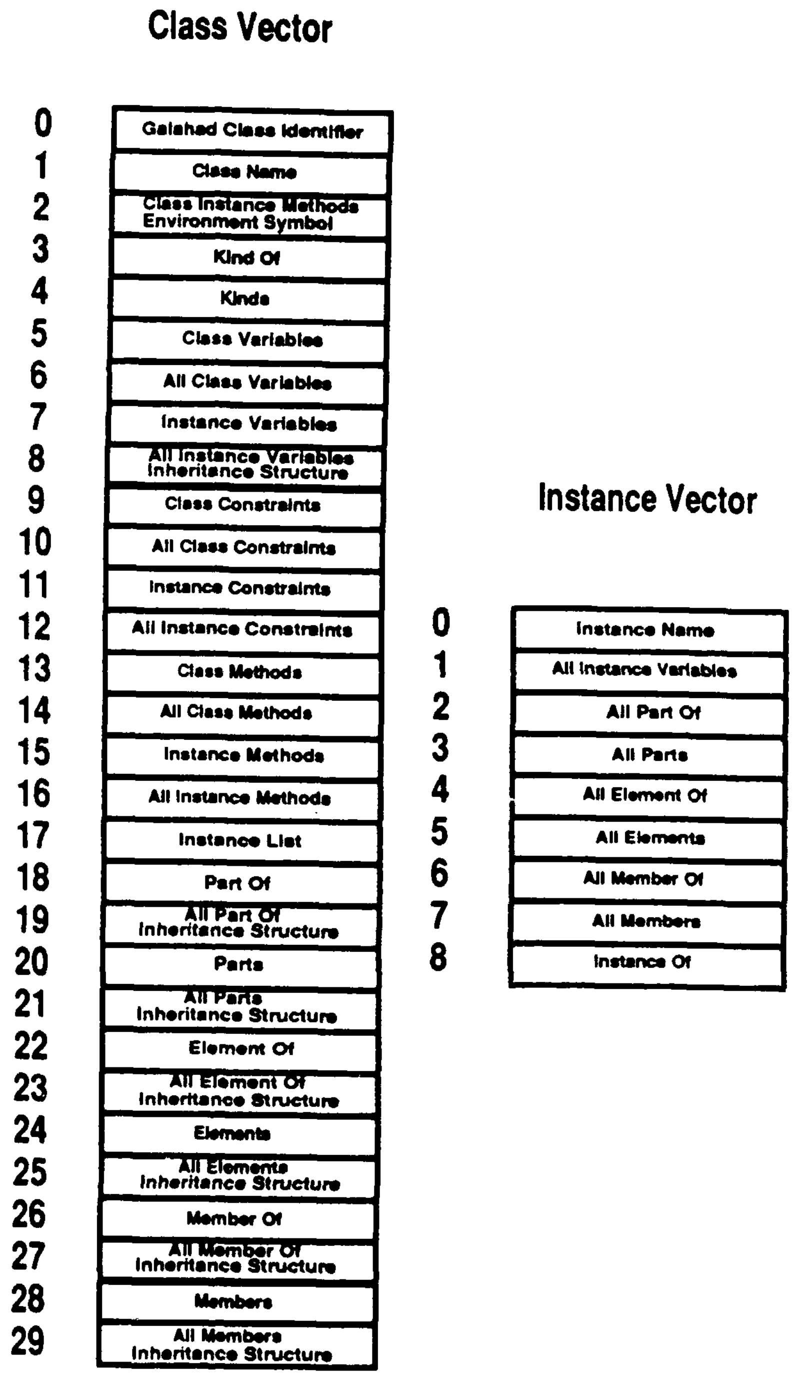


Figure 4-1. Galahad Class and Instance Vectors

functional subsystems within Galahad while METACLASS and OBJECT are actual Galahad classes. These four components share a pool of universally available support primitives. Figure 4-2 gives a graphical representation of Galahad's logical design.

SEND. SEND is responsible for all message passing that occurs within the system. This component sends messages by building and evaluating function calls in the proper Scheme environment. Details about the mechanics behind a SEND operation will be presented later in this section.

CONTROL. CONTROL monitors the user's application to ensure compliance with all model constraints. These constraints are presented along with Galahad's modeling requirements in Chapter 2 of this thesis.

To enforce some of the modeling constraints, CONTROL uses a temporary storage area, called the Scratch Pad. The Scratch Pad contains a list of those interclass links that require a "counterpart" to be considered complete. An example of a part/counterpart interclass link is the KIND-OF/KINDS relationship. If a user specifies that a SUBMARINE is a KIND-OF WARSHIP, CONTROL will prohibit adding the KIND-OF link to SUBMARINE until it has encountered a corresponding KINDS link from WARSHIP.

To enforce domain constraints, CONTROL interprets the user model in one of two modes: DELAY-COMPILE and COMPILE. DELAY-COMPILE allows the Galahad programmer to enter an entire model into the system without CONTROL's enforcement of domain constraints. This feature is crucial, especially in the modeling of interclass links where the domain of one link is a yet-to-be defined Galahad class.

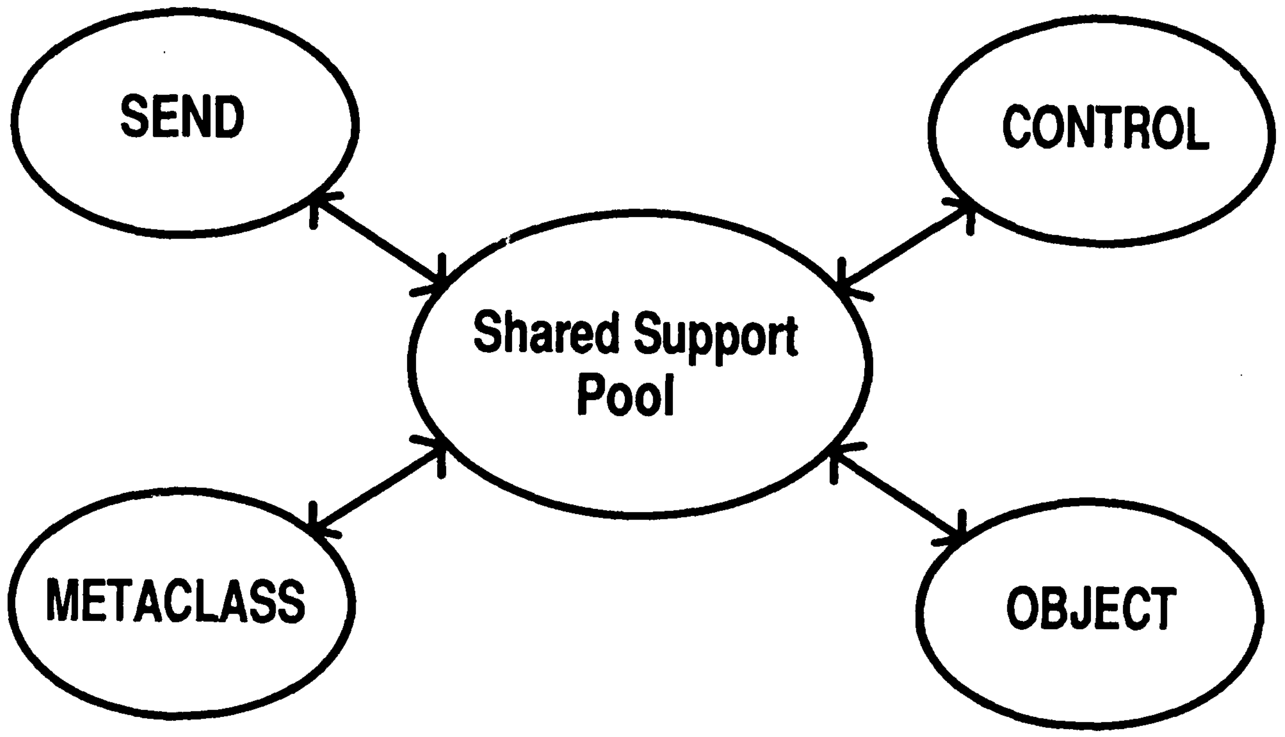


Figure 4-2. Galahad Logical Design

For example, a CONVOY may have a SHIP as one of its Members. For the reverse link, SHIPs are Members-Of CONVOYs. For domain enforcement to work, SHIP must be present in the system during the creation of CONVOY, and CONVOY must be present in the system before the creation of SHIP. This "chicken and the egg" problem is solved by CONTROL temporarily not enforcing domain constraints while in DELAY-COMPILE-MODE. Upon the user's invocation of the COMPILE command, DELAY-COMPILE-MODE is turned off and the just-entered model is examined to ensure domain compliance.

METACLASS. The METACLASS component of the system has the same purpose as METACLASS in Smalltalk. Specifically, METACLASS contains those primitives necessary to create and modify Galahad classes. Also, this component contains those routines necessary to complete the generalization inheritance structure for each class. The completion of this inheritance structure is commonly referred to as COMPILING the class.

OBJECT. Galahad's OBJECT class again follows Smalltalk's lead and fulfills the same functions as Smalltalk's OBJECT class. OBJECT in the Galahad system poses as the top-most class in the generalization inheritance hierarchy. All user-defined classes are subordinate to this class. Consequently, all user-defined classes inherit those variables and methods defined in OBJECT. Examples of methods inherited from OBJECT include the instance creation procedures and those methods reporting the contents of individual slots in the class vector.

Shared Support Pool. The shared support primitives within Galahad are not a functionally separate component within the system. Rather, they are a set of low level support routines shared by SEND, CONTROL, METACLASS, and OBJECT. The types of functions present within this shared pool are basic list and vector manipulation routines.

Galahad Environment Frame Design

Figure 4-3 shows the environment structure for Galahad after the system has initially been loaded. The reader will note that five additional frames have been added to the User Global Environment and User Initial Environment. Each of these additional environments is briefly discussed below. Unless otherwise noted, all references to a Scheme environment refer to the lowest frame of that environment.

Galahad Global Environment. This environment contains the routines in the SEND and CONTROL components of the system. It also includes all the functions in the shared support pool. The Galahad Global Environment could have been combined with Scheme's User Initial Environment; however, we chose to use a separate environment to avoid mixing Galahad and Scheme primitives at the User Initial level. This way, the boundary between Galahad and Scheme is more clearly defined.

Galahad User Environment. This environment contains symbols pointing to the METACLASS, OBJECT, and OBJECT-INST sub-environments. It also contains all class and instance symbols created by the Galahad programmer while modeling a particular application. It is, in other words, the Galahad programmer's work area.

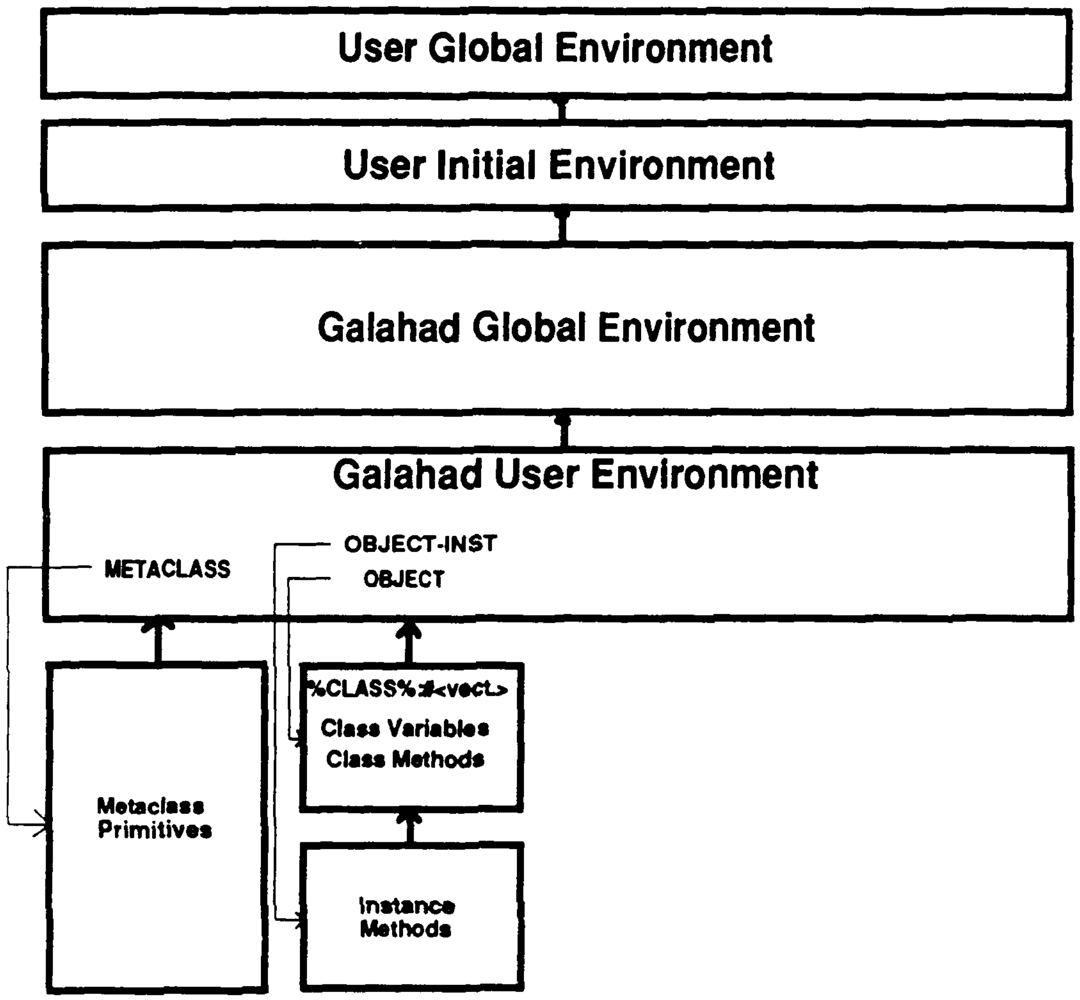


Figure 4-3. Initial Environment Frame Structure

The modeler communicates directly with this environment. The system has its own READ-EVAL-PRINT loop that evaluates in the Galahad User Environment all user-supplied expressions. This READ-EVAL-PRINT loop was adopted from Eisenberg and Abelson's book Programming in Scheme, and is presented in figure 4-4. (Eisenberg, 1988) This function is the only Galahad procedure object contained in the User Initial Environment.

The Galahad User Environment resides below the Galahad Global Environment because the global environment contains the SEND and CONTROL components of the system. The user must access these components while communicating with Galahad.

METACLASS environment. The METACLASS environment contains all METACLASS primitives used by the system. The Galahad programmer accesses METACLASS functions by following the message passing protocol. METACLASS resides beneath the Galahad User Environment and thus has access to all modeler-defined symbols in the Galahad User Environment. METACLASS has access to the shared pool, SEND, and CONTROL functions contained in the Galahad Global Environment as well.

OBJECT environment. The OBJECT environment contains all class variables and class methods associated with the OBJECT class. Also, this environment contains the symbol %CLASS% which points to the OBJECT class vector. Since OBJECT is a class, the modeler accesses OBJECT's class variables and methods by following the message passing protocol. Like METACLASS, OBJECT resides beneath the Galahad User Environment. Consequently, OBJECT has access to all symbols higher in the environment hierarchy.

```
(define start-galahad
  (lambda ()
    (letrec
      ((type-at-environment
        (lambda (passed-prompt passed-environment)
          (newline)
          (display passed-prompt)
          (let
            ((input-expression (read)))
            (writeln (eval input-expression passed-environment))
            (type-at-environment passed-prompt passed-environment)))))
      (type-at-environment
        "Galahad --> "
        (access galahad-user-environment galahad-global-environment)))))
```

Figure 4-4. Code Showing the Implementation of Galahad's READ-EVAL-PRINT Loop

OBJECT-INST environment. The OBJECT-INST environment contains all instance methods associated with the OBJECT class. Unlike OBJECT, messages are not sent directly to OBJECT-INST. Instead, OBJECT-INST poses as a parent environment for the creation of an instance environment during a SEND to an instance. Again, details behind the mechanics of a send will be presented in the next section. OBJECT-INST resides beneath OBJECT. This enables OBJECT's instance methods to have direct access to OBJECT's class variables, class methods, and other primitives higher in the environment hierarchy.

The environment structure associated with user-defined classes is very similar to the environment frames associated with OBJECT. For example, the creation of the class WAR-SHIP causes WAR-SHIP and WAR-SHIP-INST environments to be created. Figure 4-5 shows Galahad's environment structure after an arbitrary number of user classes have been defined.

The Implementation of Message Passing in Galahad

A SEND in the Galahad system is really nothing more than a fancy function call. The SEND macro is responsible for converting the send statement to an S-expression suitable for a direct invocation of the desired method. This S-expression is treated differently, depending on whether the destination of the message is either a class or an instance. If the destination is a class, the S-expression is evaluated directly in the class's class methods environment. If the destination is an instance, the process becomes more complicated because Galahad instances are not modeled as environments. The primary steps involved in sending a message to an instance are as follows:

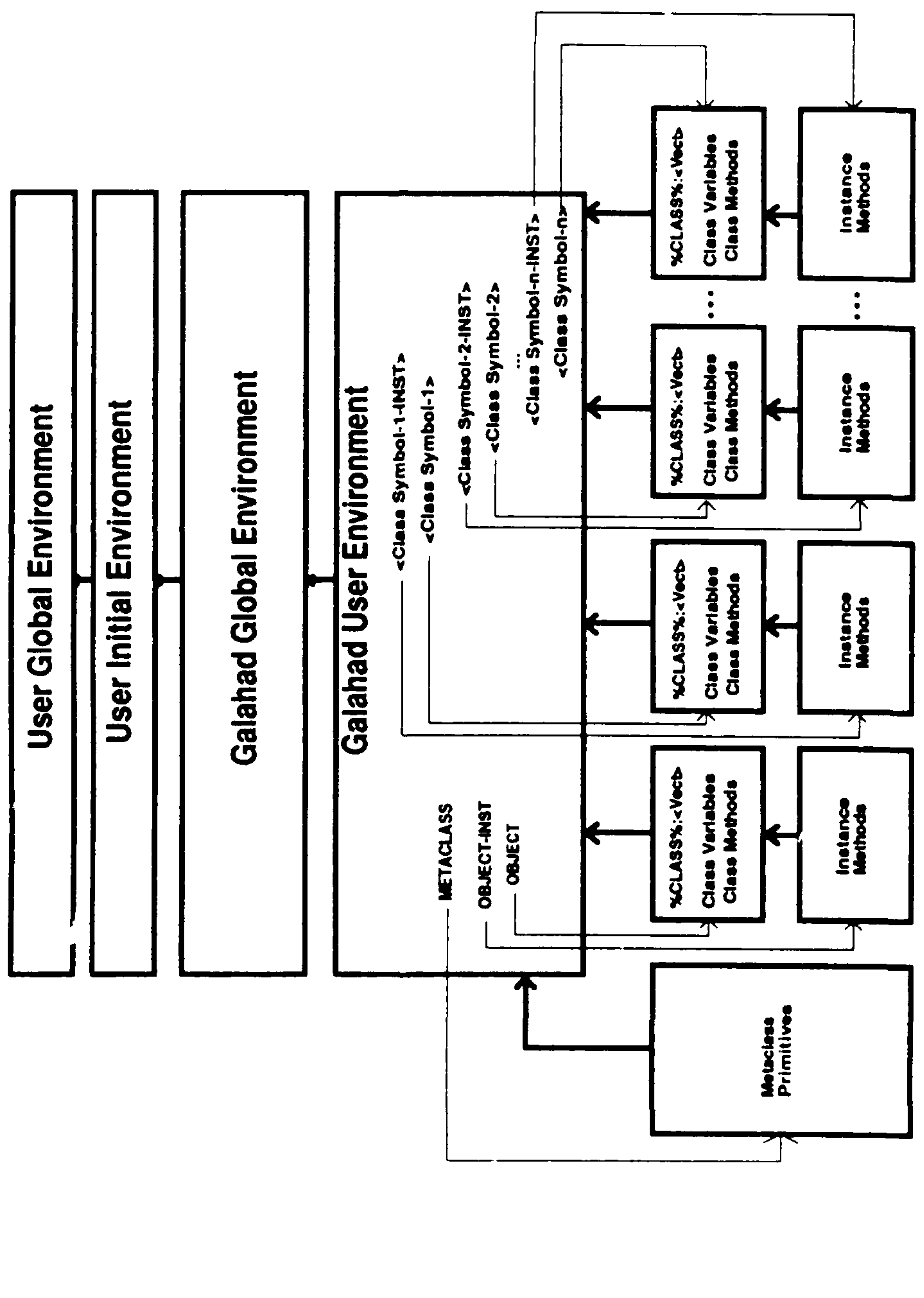


Figure 4-5. Galahad Environment Frame Structure

1. Determine which defining class of the instance is to be used for the processing of the message. This determination is a function of both the intent of the message and the set of classes which define the instance.

2. Build a list of symbol/value binding pairs for the instance based on the just determined defining class.

3. Build a temporary environment frame whose parent is the instance-methods environment of the defining class for the instance.

4. Evaluate in that environment the S-expression built by the SEND macro.

To clarify our discussion, we will examine each step as it applies to a specific example. Consider figure 4-6. This example is identical to that presented in 2-5 and is repeated here for convenience. Assume we are to send to the USS Missouri the message GET-REGISTRY. GET-REGISTRY is a method that returns the value of the instance variable Registry. Also assume the instance is bound to the symbol MISSOURI. The syntax for the send and selected portions of the example's instance and class vectors are presented in figure 4-6.

Determine defining class. The defining class for the instance can be determined in two different ways. The first is to use the optional key word DEFAULT in the SEND statement. The second is to examine the instance's Instance-Of slot and return the value(s) found there. Since the SEND statement presented in figure 4-6 has no optional DEFAULT key word, we use the second alternative. In this particular case, the contents of the Instance-Of slot is WAR-SHIP.

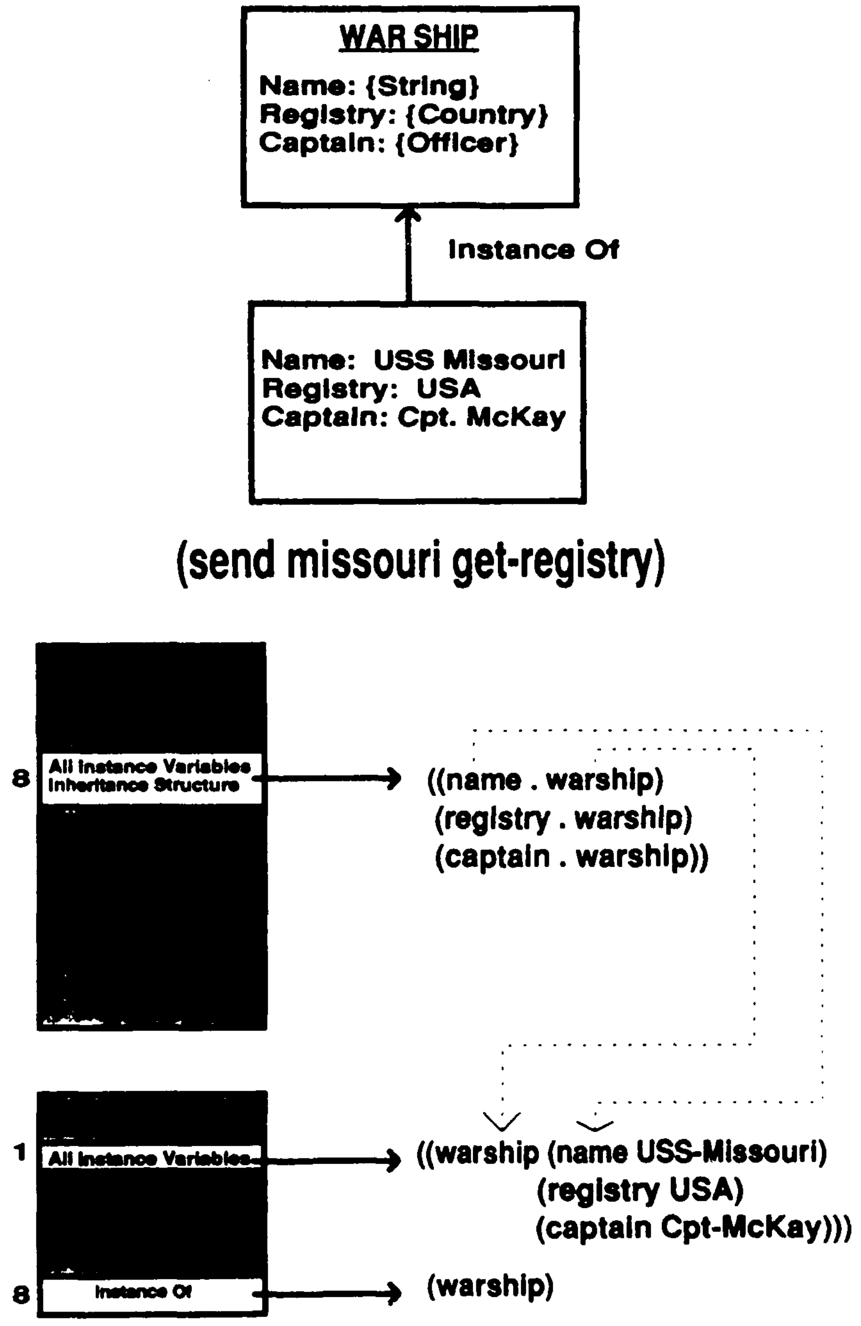


Figure 4-6. Implementation of Message Passing

Build symbol/value binding pairs. After having determined the appropriate defining class of the instance, we can now build a list containing the symbol/value binding pairs of all instance variables associated with the MISSOURI. The process is to take each variable listed in WAR-SHIP's All-Instance-Variables-Inheritance-Structure slot and retrieve the corresponding symbol/value pairs from the All-Instances-Variables position of the instance vector. This process is shown graphically by the arrows in figure 4-6. The code is presented in figure 4-7. The resulting symbol/value binding pairs for this particular example are ((name USS-Missouri) (Registry USA) (Captain Cpt-McKay))).

The reason for storing instance variables in an ASSOC list based on the defining class is due to the requirement for modeling instance lattice inheritance. With instance lattice inheritance, the All-Instance-Variables slot for an instance may contain variables with the same name. To differentiate these variables, they must be stored with their defining class.

Build sub environment. After the symbol/value binding pairs have been determined, Galahad builds a temporary sub-environment, which we refer to as the instance environment. The parent of the instance environment is the instance-methods environment of the defining class. The symbols contained in the instance environment and their associated values are derived from the symbol/value list created in the previous step. Figure 4-8 shows Galahad's environment frame structure after the instance environment has been created. The instance environment is shown as a dashed box because the environment is

```
(define-galahad-primitive build-instance-environment-iv-binding-pairs
   (lambda (passed-instance-data passed-structure)
      (let
         ((build-iv-binding-pair
            (lambda (context-pair)
               (let*
                  ((variable-name (car context-pair))
                   (defining-class-name (cdr context-pair))
                   (variable-value
                        (cadr
                           (assoc
                              variable-name
                              (cdr
                                    (assoc defining-class-name passed-instance-data))))))
                  (list variable-name '',variable-value)))))
            (map build-iv-binding-pair passed-structure))))
```

Figure 4-7. Code Showing the Construction of Environment Bindings

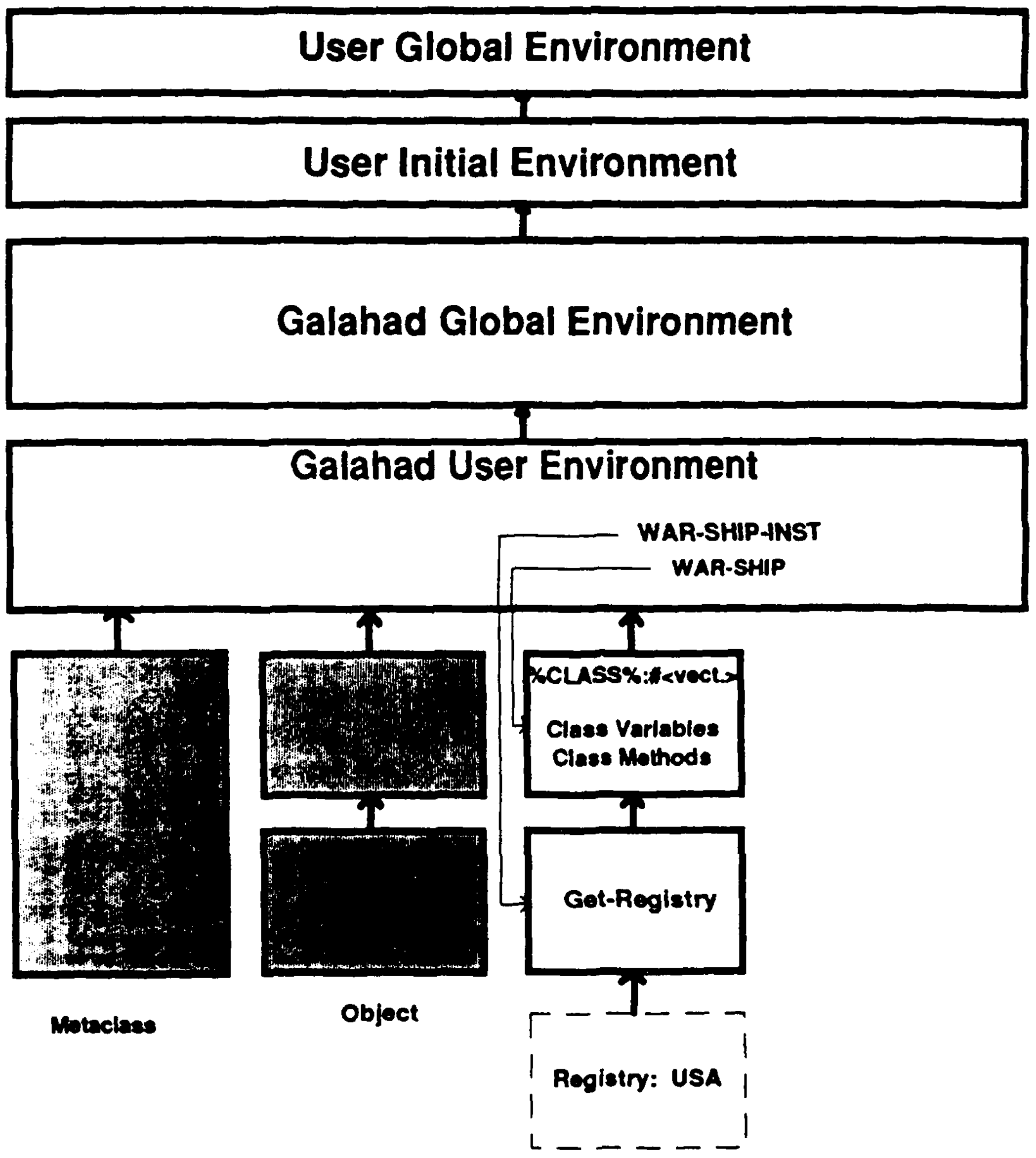


Figure 4-8. Environment Frame Structure During the Evaluation of a Method

temporary and no symbol directly points to it. The environment is discarded after the evaluation of the SEND-generated S-expression.

Evaluate function. The last step is to evaluate in the instance environment the S-expression created by the SEND macro. For this particular example, the SEND-generated S-expression is (GET-REGISTRY). Since GET-REGISTRY already exists in WAR-SHIP's instance-method environment, the evaluation proceeds smoothly and the value USA is returned.

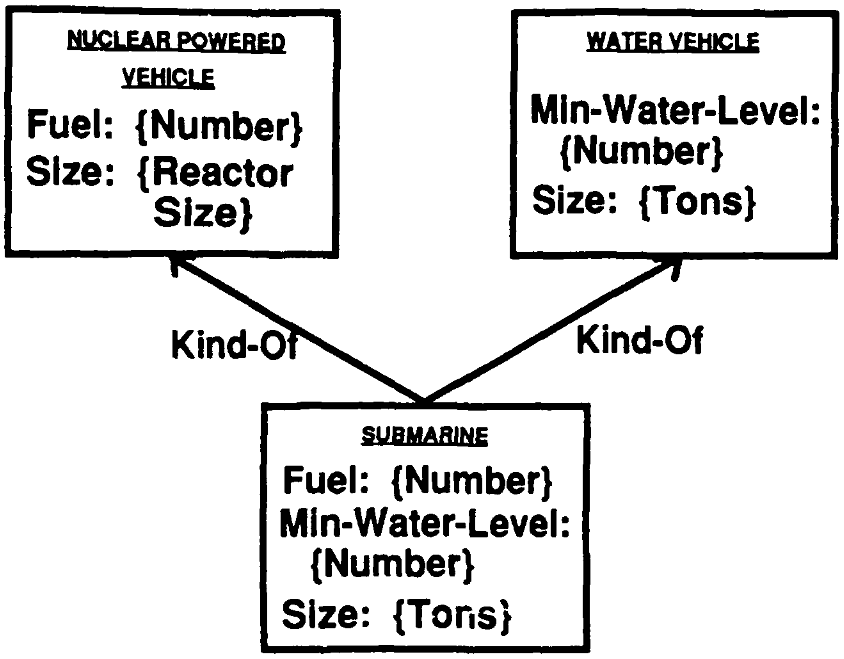
We note that a portion of Galahad's SEND code includes a small piece taken from a publicly available experimental version of SCOOPS. (Texas Instruments, 1986) This code enables us to bypass a problem associated with the lexical scoping of variables within a procedure object. For details, the reader is referred our description of DEFINE-METHOD in Appendix C, An Analysis of Texas Instrument's Implementation of SCOOPS in PC Scheme.

Modeling Requirements Implementation

We now turn our attention to the implementation of the requirements presented in Chapter 2. Where applicable, we include segments of Galahad code. Also, for convenience and continuity, we repeat the same examples presented earlier.

Lattice Inheritance

An example of a class inheritance problem is presented in figure 4-9. The figure also includes a sample Galahad statement to model the problem. In this example, we have specified that the class SUBMARINE should inherit Size from the class WATER VEHICLE. By following the conventional "depth-first up to



```
(send 'metaclass 'create-class 'submarine
        '(kind-of  (nuclear-powered-vehicle)
                   (water-vehicle (iv size))))
```

Figure 4-9. Lattice Inheritance Example (Revisited)

joins" protocol and the specific override from WATER VEHICLE, SUBMARINE will assume the instance variables Fuel, Size (from WATER VEHICLE), and Min-water-level.

Galahad builds the inheritance hierarchy during the COMPILE-CLASS process. Specifically, COMPILE-CLASS alters the All-Instance-Variables-Inheritance-Structure slot (vector position 8) for the SUBMARINE vector. The order of precedence followed by COMPILE-CLASS in assigning values to the slot is first locally defined variables, then variables identified by the user to be inherited, and finally remaining variables contained in SUBMARINE's superclasses, taken in left to right order. Of course, a fundamental directive while building the All-Instance-Variables-Inheritance-Structure is that no duplicate variable names are permitted in the just-compiled class. Figure 4-10 shows the segment of COMPILE-CLASS code that modifies the All-Instance-Variables-Inheritance-Structure position.

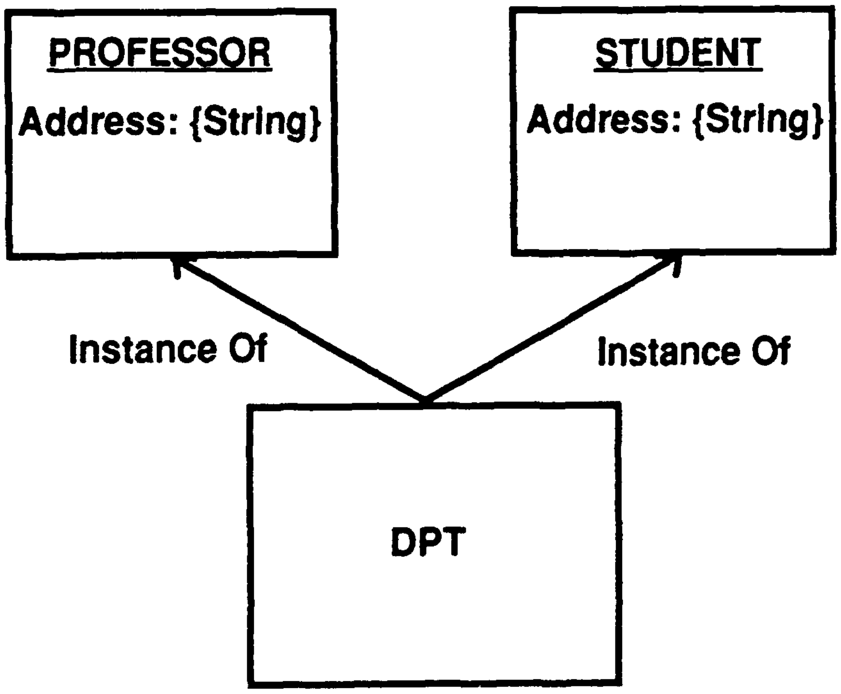
While our example centers on the inheritance of instance variables, the same type of procedure occurs with class variables as well. The only difference is that COMPILE-CLASS modifies the All-Class-Variables slot rather than the All-Instance-Variables-Inheritance-Structure position. Also, the format for the All-Class-Variables position is different due to class variables not being subjected to special requirements for instance lattice inheritance. The reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors, for details.

Instance Lattice Inheritance

An example of an instance lattice inheritance problem is presented in figure 4-11. We also include a sample Galahad statement to model the problem.

```
(define-metaclass-primitive compile-class
  (lambda (passed-class-name compile-option)
    (letrec
      ((compile-element
        (lambda (class-name)
          (let*
            ((actual-class
              (eval '(access %class% ,class-name) galahad-user-environment))
             (kind-of-structure (class-kind-of actual-class)))
            (case compile-option
              (instance-variables
                (set! (class-all-instance-variables-inheritance-structure
                        actual-class)
                      (build-inheritance-list
                        (append
                          (build-variable/defining-class-list
                            (class-instance-variables actual-class)
                            (class-class-name actual-class))
                          (extract-instance-variable-override-values
                            kind-of-structure))
                        class-all-instance-variables-inheritance-structure
                        kind-of-structure))
                (set! (class-all-instance-methods actual-class)
                      (build-inheritance-list
                        (class-instance-methods actual-class)
                        class-all-instance-methods
                        kind-of-structure))
                (eval '(set! (access
                               ,(class-class-name-inst actual-class)
                               galahad-user-environment)
                             (build-instance-environment ,actual-class))))
                        .
                        .
                        .
```

Figure 4-10. Segment of COMPILE-CLASS Code Used to Implement Lattice Inheritance



```
(send ('professor 'student) '%create-instance% 'dpt
          '(professor address ("University of Colorado"))
          '(student address ("123 Elm Street")))
```

Figure 4-11. Instance Lattice Inheritance Example (Revisited)

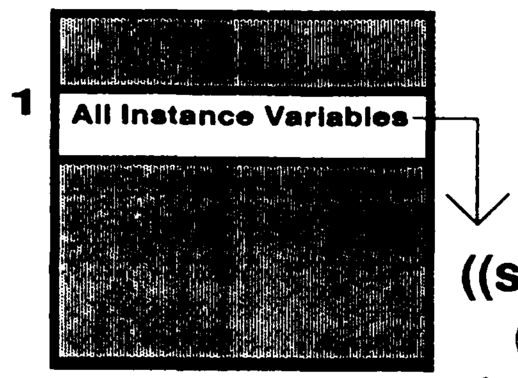
In this example, DPT is to be an instance of two classes, PROFESSOR and STUDENT. Additionally, we have assigned the instance variable Address in PROFESSOR to be "University of Colorado." STUDENT Address is "123 Elm Street."

Galahad builds the DPT instance in two distinct phases. The first phase creates the instance vector without any values assigned to any of the slots. The second phase adds the attributes of DPT's defining classes. For this particular example, we add PROFESSOR and STUDENT.

A defining class is added to an instance by adding another component to the list in the All-Instance-Variables slot of the instance vector. Components are added to the lists in the instance's instance aggregation slots as well. The CAR of each list component is either the name of the newly added defining class, or the name of a defining class's superclass. The CDR of the list contains the symbol/value pairs of variables that were locally defined for the class or class's superclass.

The reason for decomposing a defining class into its component superclasses is economy of storage. For example, if PROFESSOR and STUDENT have the superclass PERSON, and PERSON contains instance variables, then those variables need only be stored once in the vector. The CAR of the list component in this case would be PERSON. Figure 4-12 shows a completed All-Instance-Variables slot for the example presented in figure 4-11.

Evaluation of instance methods in an instance lattice situation occurs in the same manner as described in the section on the implementation of Galahad message passing. The only difference is that the SEND mechanism will process a message using all defining classes for an instance if no class has been specified by



```
((student
   (address "123 Elm Street""))
 (professor
   (address "University of Colorado")))
```

Figure 4-12. Completed All-Instance-Variables Slot for the Instance DPT

the user. The appropriate defining classes are derived by examining the Instance-Of slot of the instance vector. In the example presented in figure 4-11, the contents of this slot for the instance DPT is (PROFESSOR STUDENT).

The code presented in figure 4-13 shows the function sending messages to an instance when the instance has multiple defining classes.

Abstract Superclasses

Figure 4-14 shows an example of an abstract superclass problem and the accompanying Galahad statement. In this particular example, ENGINE has instance variables Manufacturer and Diesel. Since ENGINE has been identified as an abstract class, Galahad does not permit the instantiation of the class.

We implemented the abstract class concept by defining in OBJECT a class variable called ABSTRACT?. This variable contains a boolean value indicating whether or not Galahad should treat the class as an abstract entity. Since ABSTRACT? is in OBJECT, the variable automatically is inherited by all user-defined classes.

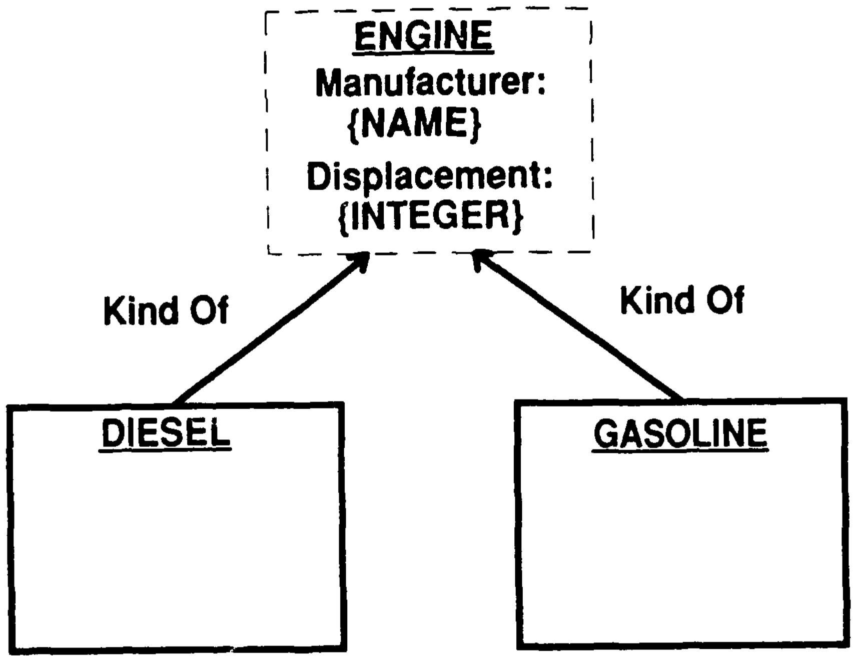
The class method %CREATE-INSTANCE% tests the value of ABSTRACT? before allowing the class to create an instance. If ABSTRACT? is true, %CREATE-INSTANCE% displays an error message to the user. Otherwise, it allows the normal creation of an instance.

Simple Classes

Because simple classes are procedural definitions, they are implemented in the system as standard Scheme procedures. The procedures reside in the

```
(define-galahad-primitive send-message-with-multiple-contexts
  (lambda (passed-function-call passed-destination passed-contexts)
    (if (null? passed-contexts)
        *the-non-printing-object*
        (let*
          ((next-context-class
              (eval
                `(access %class% ,(car passed-contexts))
                galahad-user-environment))
           (next-context-instance-environment
              (eval
                (class-class-name-inst next-context-class)
                galahad-user-environment))
           (results-of-next-send
              (send-message
                passed-function-call
                (build-sub-environment
                  (build-instance-environment-binding-pairs
                    passed-destination
                    next-context-class)
                next-context-instance-environment))))
          (if (not (eq? results-of-next-send '%undefined-method%))
              (writeln
                (car passed-function-call)
                " evaluated in "
                (car passed-contexts)
                " context:  "
                results-of-next-send))
          (send-message-with-multiple-contexts
            passed-function-call
            passed-destination
            (cdr passed-contexts))))))
```

Figure 4-13. Code Showing Implementation of Message Sending when an Instance has more than One Defining Class



```
(send 'metaclass 'create-class 'engine
   '(kinds diesel gasoline)
   '(iv (manufacturer name (#!unassigned) (1 1))
        (displacement integer (#!unassigned) (1 1)))
   '(abstract))
```

Figure 4-14. Abstract Superclasses Example (Revisited)

Galahad User Environment and are invoked just like any other Scheme function. Simple classes were not designed to respond to a message.

When creating a user-defined simple class, Galahad modifies the class name by adding a "?" to the end of the name symbol. For example, the simple class NAME is modified to become NAME?. This was done to be consistent with Scheme's naming conventions for its data typing functions.

Domain/Cardinality Constraint Enforcement.

Stored with each class, instance, and instance aggregation variable is information concerning domain and cardinality constraints. For example, the instance variable Captain in class WAR-SHIP may have a domain of OFFICER and a cardinality of (1 1). The specific format for the storage of this information is presented in Appendix A.

The system enforces a variable's domain and cardinality limits each time the variable is updated. In Galahad, all updates to a variable occur through one function, %SET-GALAHAD%. This function, which specifically shows constraint enforcement, is shown in figure 4-15. Galahad also uses %SET-GALAHAD% to establish a variable's default and initial values.

The cardinality of variables is enforced by performing a test on the length of the value list associated with each variable. For example, assume the instance variable Registry in class WAR-SHIP has the value (USA LIBERIA). The cardinality for this particular variable would be at least two.

Domain enforcement primitives are created each time the user creates a Galahad class. The names of these primitives follow the same pattern used by Scheme data typing functions and Galahad simple classes. For example, when the

```
(send 'metaclass 'create-class-method 'object '%set-galahad%
  '(lambda ((passed-self) (passed-variable-name) (passed-new-value-list))
     (let*
       ((variable-structure
          (assoc passed-variable-name
                 (append
                   (%get-all-class-variables% self)
                   (%get-all-instance-variables% self))))
       (variable-domain (cadr variable-structure))
       (variable-cardinality (cadddr variable-structure))
       (variable-values
          (if (equal? passed-new-value-list '(#!unassigned))
              #!unassigned
              passed-new-value-list))
         (constraint-check-list
           (append
             (test-value-list-for-domain-constraint
               variable-values
               variable-domain
               '%set-galahad%
               (if (not (unbound? %instance%))
                    (get-instance-name self)
                    (%get-class-name% self))
               passed-variable-name)
             (list
               (if (invalid-cardinality?
                        (length variable-values)
                        variable-cardinality)
                      (galahad-error
                        2
                        '%set-galahad%
                        variable-values
                        variable-cardinality
                        passed-variable-name
                        (if (not (unbound? %instance%))
                             (get-instance-name)
                             (%get-class-name%))))))))
      (if (constraints-satisified? constraint-check-list)
          (eval
            '(begin
               (set! ,passed-variable-name ',variable-values)
               (if (not (unbound? %instance%))
                    (set!
                      (instance-all-iv %instance%)
                      (%build-new-instance-data-list%
                        (cdr
                          (assoc
                            ',passed-variable-name
                            (%get-all-instance-variables-inheritance-structure%
                              self)))
                        ',passed-variable-name
                        ',variable-values
                        (instance-all-iv %instance%))))
               (list ',passed-variable-name ',variable-values)))
          *the-non-printing-object*)))
  'nocompile)
```

Figure 4-15. %SET-GALAHAD%

class STUDENT is built, a function is created called STUDENT?. STUDENT? is the primitive that tests class membership within the STUDENT class.

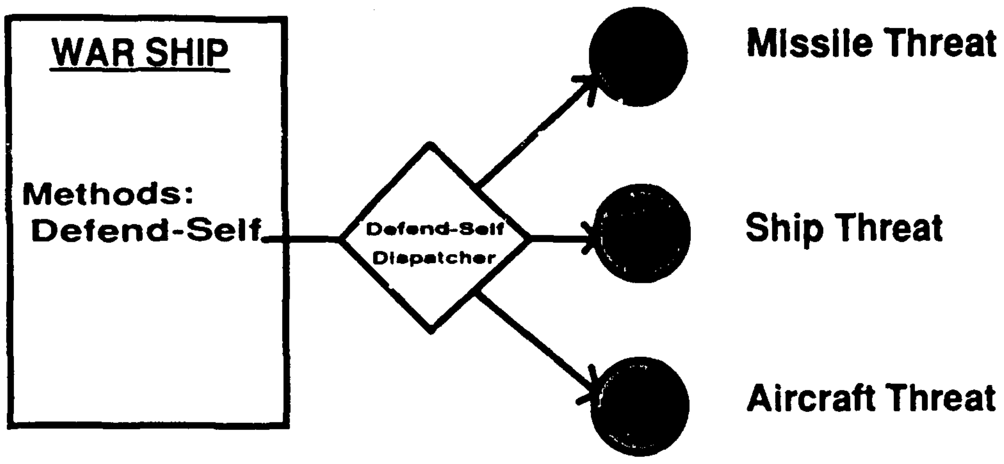
Specialized Methods

An example of a situation requiring specialized methods is presented in figure 4-16. We also include sample Galahad statements to model the instance methods which specialize on the Missile threat. For this particular case, these specialized methods directly call functions responsible for firing various weapon systems.

Galahad models specialized methods in a manner that is very similar to the graphical representation of the example. Specifically, located in WAR-SHIP's class methods environment are two symbols associated with the Defend-Self method: DEFEND-SELF and DEFEND-SELF-DISPATCH-TABLE. The contents of each symbol is shown in figure 4-17. For clarity, DEFEND-SELF is shown in source code format. The source code was generated by the Galahad routine CREATE-INSTANCE-METHOD.

DEFEND-SELF is the dispatching procedure that calls the appropriate specialized method based on the data type of the parameters passed to the dispatching routine. DEFEND-SELF uses DEFEND-SELF-DISPATCH-TABLE to determine the proper procedure to call.

Each sublist contained in the dispatch table corresponds to one specialized method. A sublist consists of names of data typing functions and a pointer to the specialized method. The dispatching routine uses the typing functions to determine whether the specialized method contained in the sublist is the appropriate routine to call.



```
(send 'metaclass 'create-instance-method 'warship
        'defend-self
        '(lambda ((threat missile))
              (fire-phalyx-guns)))

(send 'metaclass 'create-instance-method 'warship
        'defend-self
        '(lambda ((threat ship))
              (fire-harpoon-missile)))

(send 'metaclass 'create-instance-method 'warship
        'defend-self
        '(lambda ((threat aircraft))
              (fire-sparrow-missile)))
```

Figure 4-16. Specialized Methods Example (Revisited)

DEFEND-SELF:

```
(lambda (threat)
        (eval
                (append
                        (retrieve-proper-method-for-argument-types
                                defend-self-dispatch-table
                                (append
                                        (list threat)
                                        ()))
                        (build-list-of-quoted-objects
                                (append
                                        (list threat)
                                        ()))))))))
```

DEFEND-SELF-DISPATCH-TABLE:

```
((missile? #<procedure>)
 (ship? #<procedure>)
 (aircraft? #<procedure>))
```

Figure 4-17. Code Showing the Implementation of Specialized Methods

For example, assume the method DEFEND-SELF is called with a parameter of MISSILE-1. MISSILE-1 is an instance of the class MISSILE. The dispatching procedure (DEFEND-SELF) examines each sublist in DEFEND-SELF-DISPATCH-TABLE to search for the specialized method. In this particular case, the first sublist would point to the proper specialized method because MISSILE-1 would pass the MISSILE? class membership test. Had the threat been SHIP-1 instead, SHIP-1 would fail the MISSILE? test, but would pass the SHIP? test. For SHIP-1, the specialized method associated with a SHIP threat would be called.

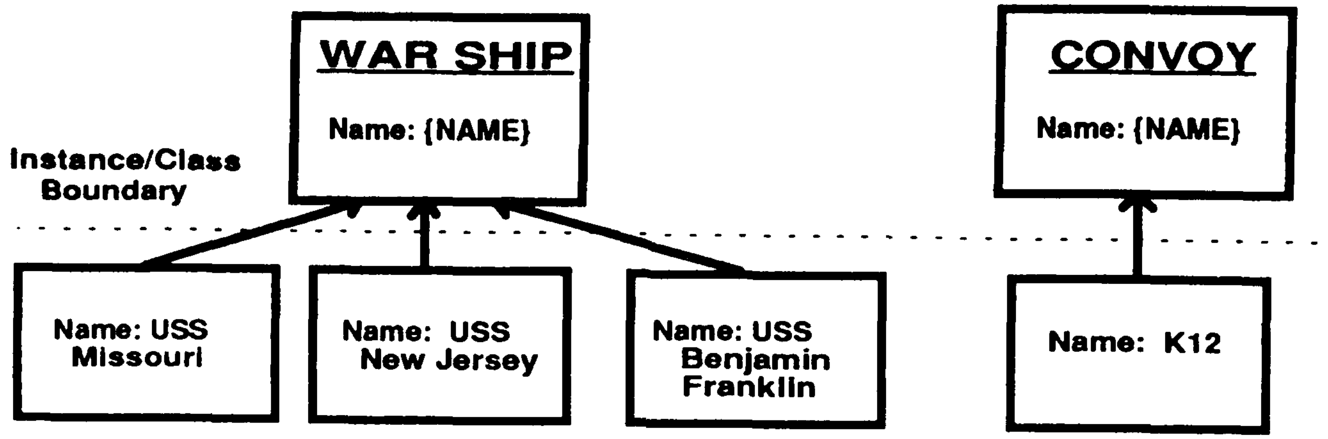
Instance Aggregation

An example of Member instance aggregation is presented in figure 4-18. As we have done with our other examples, we include sample Galahad statements. These statements build the Members/Member-Of connections between the classes WAR-SHIP and CONVOY.

For the implementation of instance aggregation, we treated the Part, Element, and Member aggregation concepts as though they were instance variables with a few notable exceptions. The exceptions are as follows:

1. Part, Element, and Member have their own individual slots in Galahad class and instance vectors.

2. Galahad monitors the user's model to ensure the programmer has included forward and reverse links on an instance aggregation connection. For example, if the class WAR-SHIP is given a Member-Of link to CONVOY, the system waits for the corresponding Members link



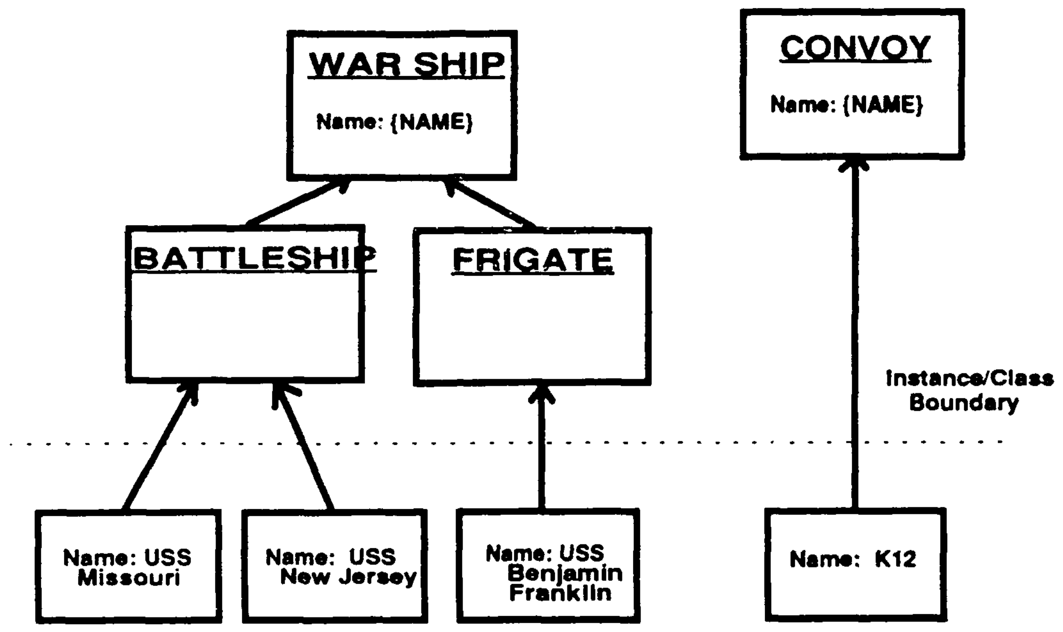
```
(send 'metaclass 'add-members 'convoy
    '((warship (1 n)))
(send 'metaclass 'add-member-of 'warship
    '((convoy (0 1)))
```

Figure 4-18. Instance Aggregation Example (Revisited)

from CONVOY to WAR-SHIP before altering either the CONVOY or WAR-SHIP class. This monitoring of the user's model was implemented using the Scratch Pad concept described earlier.

3. Instance aggregation connections defined at the class level do not require a default value. Instance aggregation connections need a specification for only cardinality constraint enforcement.

4. Instance aggregations connections stored at the instance level have the capability for storing the defining class of an instance connection. For example, assume the example shown in figure 4-18 has been modified to become figure 4-19. In this particular case, we still have the Member-Of/Members connection between WAR-SHIP and CONVOY; however, the individual instances are now members of subclasses of WAR-SHIP instead of WAR-SHIP itself. The Member-Of/Members connection still applies to the Missouri, New Jersey, and Benjamin Franklin; but Galahad must account for the Missouri and New Jersey being battleships and the Benjamin Franklin being a frigate. To do this, the system stores in the All-Members slot of the instance K12 the list shown in figure 4-19.



All-Members-Slot of K12

```
((war-ship (battleship missouri new-jersey)
           (frigate benjamin-franklin)))
```

Figure 4-19. Instance Aggregation: Differentiation Among Different Subclasses

CHAPTER V

DIRECTIONS FOR FUTURE RESEARCH AND CONCLUSION

As with any research project, our implementation of Galahad has uncovered many areas for further study. Also, there are several small coding problems that still need to be addressed. The purpose of this chapter is to identify these directions for future coding and research. For those areas requiring further code writing, we include a brief discussion on the general approach the programmer should take.

We begin this chapter by first describing the minor programming tasks that remain to be done. We then discuss directions for additional functionality to be added to the system. We conclude by offering our insights for the future role of Galahad at the University of Colorado.

Minor Programming Tasks

This section describes the programming details we purposely bypassed in our attempt to achieve the greatest functionality over the shortest period of time. While these details are not crucial for the demonstrated success of Galahad, they are important if the system is ever to become a commercially viable product.

Modification of Instance Aggregation Specifications to Include Specific Support for the Logical Operators AND, OR, and XOR.

Galahad allows programmers to model instance aggregation concepts; however, it does not provide explicit support for the logical operators *AND*, *OR*, and *XOR*. An example of a situation requiring a logical operator would be a CONVOY having WAR-SHIPs, FREIGHTERs, *OR* TANKERs as valid Members. A Galahad programmer can model *AND* and *OR* concepts in the current system by using the cardinality associated wit!. an instance aggregation specification; however, explicit support for these operators would make the relationship among aggregations more clear.

One way to implement *AND* and *OR* connections on instance aggregations is to test for the appropriate key word and then internally represent within the class vector *AND* specifications as having a lower cardinality of 1 and *OR*s with a lower cardinality of 0. Unfortunately, this does not work for the *XOR* operator. To implement *XOR*, the programmer must modify the instance-level instance aggregation routines.

Modification of Constraint Enforcement to Include Removal of Partially Completed Transactions

The current version of Galahad detects all constraints as specified by the requirements presented in Chapter 2; however, the CREATE-CLASS and %CREATE-INSTANCE% routines must be altered to reverse those definitions when a constraint has been violated. For example, assume the user creates the instance USS-Missouri, but fails to specify a value for the required instance variable Captain. The current system will detect the omission of this required instance variable, but it will not delete the just-created instance USS-Missouri.

Similarly, if there is a constraint violation during the creation of a class, Galahad will detect the violation, but it will not remove the partially specified class from the system.

To implement the removal of partially completed transactions, the programmer must first develop the DELETE-CLASS-VARIABLES, DELETE-INSTANCE-VARIABLES, DELETE-CLASS-FROM-INSTANCE, and DELETE-CLASS primitives. After developing this functionality, the programmer can then modify CREATE-CLASS and CREATE-INSTANCE to use these primitives if a constraint violation has been detected. The system knows whenever a constraint has been violated because the routines detecting a violation always return the Scheme object *THE-NON-PRINTING-OBJECT*.

Modification of COMPILE-CLASS to Enable Changes in Class Structure to be Propagated to Instances.

COMPILE-CLASS in the current system modifies the structure of only Galahad classes. COMPILE-CLASS needs to be altered so that a class's instances reflect any structural changes made to their defining class. For example, assume the class WAR-SHIP has the instance USS-Missouri. If we add a new instance variable to WAR-SHIP, such as Crew-Size, this addition should be reflected in the instance USS-Missouri.

For COMPILE-CLASS to reflect this class-to-instance "link," the programmer must write code that compares all the slots of a class vector with the corresponding positions in an instance vector. If any additions are needed to an instance's All-Instance-Variables slot, then COMPILE-CLASS must make the necessary modifications. If the instance contains an attribute that no longer applies to its defining class, the instance should be flagged as "invalid."

Additionally, this code must re-check the domain and cardinality constraints for all values stored with the instance. This is necessary because a domain or cardinality specification might be altered in a class and this change needs to be reflected in the class's instances.

Modification of %ADD-CLASS-TO-INSTANCE% to Test for an Abstract Class

The current version of Galahad tests to see whether a class is abstract at instance creation time. If a class has been identified by the user to be an abstract entity, then the system prohibits the creation of the instance. Unfortunately, this test for abstraction should occur when a class is added to an instance rather than when the instance is created. For example, assume we create the instance DPT which is an instance of both the PROFESSOR and STUDENT classes. Also assume the STUDENT class is defined to be an abstract entity. The current version of the system allows STUDENT to be added to DPT. This should not be permitted.

To implement this, the programmer must add an ABSTRACT? test to the routine %ADD-CLASS-TO-INSTANCE%. If a class has been specified by the user as being ABSTRACT, then %ADD-CLASS-TO-INSTANCE% must prohibit the addition of the class to the instance. Also, the programmer may want to leave the ABSTRACT? test in %CREATE-INSTANCE%. This way, an instance vector will not be created if the first class to be added to an instance is labeled as being ABSTRACT.

Modification of DELETE-CLASS-METHOD and DELETE-INSTANCE-METHOD to Account for Specialized Methods

The current version of the system deletes class and instance methods by entirely removing both the method's dispatching function and the dispatch table. These functions must be altered so they remove only an entry from the dispatch table. If the dispatch table is empty after removal of an entry, the Galahad function can then remove the dispatching routine.

To implement, the programmer must write a low-level routine that deletes individual entries from the dispatch table. The DELETE-CLASS-METHOD and DELETE-INSTANCE-METHOD routines must also be able to detect when a dispatch table is empty. If the table is empty, then the dispatch function and the table must be removed from the appropriate slots in the class vector. Low level vector manipulation routines are already in place to remove symbol/procedure pairs from a class's Class-Methods and Instance-Methods slots.

Modification of Class Inheritance to Include Selective Overrides for Class and Instance Methods.

The current Galahad system allows the programmer to override the "depth-first up to joins" lattice inheritance rule for only class and instance variables. The capability is needed for a modeler to select specific class and instance methods as well.

To implement, the programmer must make minor changes to the COMPILE-CLASS function. The changes required would be very similar to those used by COMPILE-CLASS to implement class and instance variable overrides.

Additional Functionality

This section describes those areas of Galahad system where additional functionality still needs to be added. Some of the functions we present below were not implemented because they fell victim to other, more pressing details. Other functions we describe were not implemented because they involve theoretical issues we have not yet completely resolved. In those cases requiring further conceptualization, we identify the issues under investigation.

The Role Concept for Viewing Data

An issue related to the instance lattice inheritance concept is the interpretation of queries in terms of "roles" rather than contexts. For conceptual modeling, the term "role" implies semantics different from our use of the word "context." We have been referring to contexts in the sense of the defining class of an instance. Roles, on the other hand, imply something different. Unfortunately, the term has different interpretations based on one's individual perspective. Because of these different definitions for a "role", we have not come up with our particular interpretation for implementation in Galahad. This is one area that requires more conceptualization before being crystalized in code.

Before, After, and Around Methods

The Common LISP Object System (CLOS) provides its programmers with the capability for adding before, after, and [illegible]d-methods to a class or instance primary method. (Keene, 1989) Before-methods and after-methods provide the programmer with added flexibility in describing the behavior of an object. Before-methods are automatically called before the primary method is invoked. After-methods are called after the primary method has finished. Around-methods can be viewed as "supervisory" methods because they can alter the sequence of calls to before, primary, and after-methods.

An illustration of before, after, and around-methods can be shown with our WAR-SHIP example and the primary method Defend-Self. An around-method to Defend-Self may (and probably should) be to verify that an approaching threat is hostile. If the threat is hostile, the around-method would then permit the before, primary, and after-methods to be called. Otherwise, the around method would bypass the standard calling sequence and return WAR-SHIP to the status quo. An example of a before-method for Defend-Self may be to Track-Target. An example of an after-method may be to Verify-Threat-Destruction.

To implement these concepts, the programmer probably will have to overhaul the method evaluation code in the current version of Galahad. Mechanisms need to be added that will provided explicit support for the programmer to specify the before, after, and around-methods to be tied to the primary method. Also, since a subclass may call before and after-methods of one of its superclasses, the programmer must ensure a "Send to Super" mechanism is in place.

Dependent Classes and Dependent Instances

The interclass existence dependency modeling requirement presented in Chapter 2 was the only coding specification not implemented in this version of Galahad. To implement inter-class and inter-instance dependencies, the programmer must develop two new "links", complete with reverse connections, between classes.

The first link is a CLASS-DEPENDENT-ON/CLASS-DEPENDENTS connection which identifies a Galahad class as being existent dependent on another class. To implement this link, the programmer should parallel the code used to implement the KIND-OF/KINDS connections. The programmer must also alter the DELETE-CLASS routine so that the function also deletes all classes indicated by the CLASS-DEPENDENTS connection.

The second link is a INSTANCE-DEPENDENT-ON/INSTANCE-DEPENDENTS connection between instances. The same concepts apply as described above; however, they apply only to instances.

The need for these two separate links is due to those situations where one instance is dependent on another, but their defining classes are not. For example, consider an object-oriented database modeling EMPLOYEE as a subclass of PERSON. Also assume EMPLOYEE has an instance variable Dependents whose domain is PERSON. Obviously the instances of the dependents of an employee are existent dependent on the employee. However, the class PERSON is not existent dependent on EMPLOYEE.

Send to Super

The current system has no standardized mechanism for a class to send messages to its immediate parent(s). This capability is crucial if a Galahad class has overridden a superclass method and it needs access to the overridden method. For example, consider our WAR-SHIP example where BATTLESHIP is a subclass of WAR-SHIP. Assume the Defend-Self method for BATTLESHIP has overridden Defend-Self in WAR-SHIP. Also assume BATTLESHIP does not have a specialization for defense against a SUBMARINE attack; however, the

WAR-SHIP method does. BATTLESHIP needs to send a message to its superclass, WAR-SHIP, to Defend-Self.

To implement the "Send to Super" concept, the programmer must alter the SEND macro so that the key word SUPER automatically translates to the class's immediate superclass. For example, if BATTLESHIP issues *(send super defend-self)*, SEND must translate this to *(send WAR-SHIP defend-self)*. This translation can be done by accessing the Kind-Of slot of the class sending the message. To account for lattice inheritance, assume "Send to Super" means the first listed superclass in the class's KIND-OF slot. If the user needs to access another superclass in the lattice, then probably the best solution would be to require the user to specify the exact super class desired. That is, the specific name of the superclass will be used in place of the SUPER key word in the message.

DELETE-CLASS

Galahad allows only for the creation of classes. No DELETE-CLASS capability exists. The DELETE-CLASS function is required if the Galahad programmer is to be spared from having to reset the system and completely reload the model for each minor change made to a Galahad class.

To implement the DELETE-CLASS function, the programmer needs to unbind the symbols pointing to the class's class-methods and instance-methods environment. Also, code must be written to verify that all inter-class connections associated with the deleted class are either properly deleted or labeled as being "invalid." This specifically includes all KIND-OF/KINDS, MEMBER-OF/MEMBERS, ELEMENT-OF/ELEMENTS, and PART-OF/PARTS links.

Additionally, instance and class variables whose domain is the now deleted class must also be labeled as being "invalid."

DELETE-CLASS-FROM-INSTANCE

The current system allows a class only to be added to an instance. The inverse function has never been implemented. The DELETE-CLASS-FROM-INSTANCE function is required if Galahad is to delete a class and propagate the class deletion among all the class's instances.

To implement, the programmer must first decompose the class identified for deletion into its component superclasses. This expansion is required because an instance vector stores all variable and aggregation values by their locally defined classes. For clarification, see our discussion on Instance Lattice Inheritance in Chapter 4.

After decomposing a class, the programmer must write code that reads each component of the decomposition and removes all references of the component from all slots of the instance vector.

Particular attention must be made to ensure no needed class component is accidently removed. For example, assume PROFESSOR and STUDENT are the defining classes for the instance DPT. Also, assume these classes are subclasses of PERSON. To remove the PROFESSOR class from DPT, we first decompose PROFESSOR into PROFESSOR, PERSON, and OBJECT. We must not remove the PERSON and OBJECT components from DPT because they are still required by the STUDENT class.

DELETE-CLASS-VARIABLES/DELETE-INSTANCE-VARIABLES

The current version of Galahad has primitives to add new class and instance variables to a class. Unfortunately, we did not build commands to delete variables from an existing class.

To provide this functionality, the programmer can parallel the code used to create ADD-CLASS-VARIABLES and ADD-INSTANCE-VARIABLES. The main difference is to replace the low level primitives that add a new variable to a class vector slot with those that delete a variable from a slot. The programmer must be aware that when deleting a class/instance variable, he/she must also remove the GET/SET methods that went with the variable. Also, after a variable has been deleted, the class must be recompiled to ensure the change is propagated throughout the hierarchy.

SET-CLASS-VARIABLE-DOMAIN/SET-INSTANCE-VARIABLE-DOMAIN

The current system does not allow the Galahad programmer to alter the domain of an existing class or instance variable. Instead, the modeler is forced to redefine the entire Galahad class.

To implement the above statements, the programmer needs to write a routine that modifies the second position of a class or instance variable's specification quadruplet. For example, assume the instance variable Registry in WAR-SHIP is stored in the WAR-SHIP vector as *(Registry COUNTRY (#!UNASSIGNED) (1 1))*. This routine must be able to change COUNTRY to another value. The programmer must also write code to verify the variable's default value still passes the membership test for the newly specified domain. Also, if the domain for a class variable has been modified, the code must re-check the current value of the class variable to ensure domain compliance as well. Finally, the class needs to be recompiled to propagate the change throughout the class and instance hierarchies.

SET-CLASS-VARIABLE-CARDINALITY/SET-INSTANCE-VARIABLE-CARDINALITY

The current version of the system does not allow the Galahad programmer to alter the cardinality of a currently existing class or instance variable. Instead, the modeler is forced to redefine the entire Galahad class.

To implement the above Galahad statements, the programmer needs to write a routine that performs functions similar to those described in the previous section. The only difference is the code must alter a quadruplet's fourth position instead of its second.

DESCRIBE

Galahad does not have any provision for the modeler to print a user-readable description of Galahad classes and instances. The closest commands are %GET-CLASS% and GET-INSTANCE which returns non-edited versions of a class or instance vector.

To implement a Galahad DESCRIBE statement, the programmer can write code that examines each individual slot of a class or instance vector and pretty prints the result. The programmer may want to decompose variable and aggregation specifications into their domain, default value, and cardinality components. This way, these individual components can be labeled as they are printed.

CONCLUSION

Beyond doubt, Galahad has proven to be a smashing success. Not only has it demonstrated the feasibility of Professors Monarchi and Tegarden's conceptual modeling ideas, but also it has provided us with a solid framework for continued research. Another unexpected, but delightful, benefit of our work has been the perfection of our ideas due to rigors of computer programming. By translating the Galahad specification into a formal computer model, we were forced to fill in many details that initially were overlooked.

Admittedly, much more work is needed. Galahad is currently a prototypical system; however, over the past eight months we have laid a solid conceptual and program foundation for the continued advancement of the Galahad design. The system, while designed in an prototypical environment, was engineered for maintainability.

BIBLIOGRAPHY

1. Abelson, H., G. J. Sussman, J. Sussman (1985) *Structure and Interpretation of Computer Programs*, Cambridge, Massachusetts: The MIT Press, McGraw-Hill.

2. Alagic, S. (1986) *Relational Database Technology*, Springer Verlag, New York.

3. Bobrow, D. G. and M. Stefik (1983) *The Loops Manual*, Palo Alto, California: Xerox Corporation.

4. Booch, G. (February 1986) "Object-Oriented Development", *IEEE Transactions on Software Engineering*, SE-12(2), pp. 211-221.

5. Borgida, A., S. Greenspan, J. Mylopoulos (April 1985) "Knowledge Representation as the Basis for Requirements Specifications", *IEEE Computer*, 18(4), pp. 82-90.

6. Chen, P. P. (March 1976) "The Entity-Relationship Model - Toward a Unified View of Data", *ACM Transactions on Database Systems*, 1(1), pp. 9-37.

7. Control Data Corporation (1969) *SIMULA General Information Manual*, Palo Alto, California: Control Data Corporation.

8. Control Data Corporation (1975) *SIMULA Version 1 Reference Manual*, Sunnyvale, California: Control Data Corporation.

9. Dahl, O.J. and Nygaard, K. (1966) "SIMULA - An ALGOL-Based Simulation Language", *Communications of the ACM*, 9, pp. 671-678.

10. Dybvig, R. K. (1987) *The Scheme Programming Language*, Englewood Cliffs, New Jersey: Prentice-Hall.

11. Eisenberg, M. (1988) *Programming in Scheme*, Ed. H. Abelson, Redwood City, California: The Scientific Press.

12. Gibbs, S. J. (1985) "Conceptual Modelling and Office Information Systems", in *Office Automation: Concepts and Tools*, ed. D. Tsichritzis, Springer Verlang, New York.

13. Goldberg, A. and Robson, D. (1983) *Smalltalk-80: The Language and its Implementation*, Reading, Massachusetts: Addison-Wesley.

14. Hammer, M. and D. McLeod (1978) "The Semantic Data Model: A Modelling Mechanism for Database Applications" *Proceedings 1978 ACM SIGMOD International Conference on the Management of Data*, Austin, Texas.

15. Hammer, M. and D. McLeod (September 1981) "Database Description with SDM: A Semantic Database Model", *ACM TODS*, 6(3), pp. 351-386.

16. Keene, S. E. (1989) *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Reading, Massachusetts: Addison-Wesley.

17. Ledbetter, L. and B. Cox (June 1985) "Software-ICs", *Byte*, 15(6), pp. 307-315.

18. Monarchi, D. E., D. P. Tegarden, M. M. Nickson (1988) "The Representation and Implementation of Aggregation Hierarchies in an Object Oriented Language" *Proceedings ORSA/TIMS*, Denver, Colorado.

19. Moon, D. A., (1986) "Object-Oriented Programming with Flavors" *Proceedings ACM 1986 OOPSLA Conference*, pp. 1-8.

20. Pascoe, G. A. (August 1986) "Elements of Object-Oriented Programming", *Byte*, 16(8), pp. 307-316.

21. Pressman, R. S. (1987) *Software Engineering: A Practitioner's Approach*, New York, New York: McGraw-Hill.

22. Schmucker, K. J. (August 1986) "Object-Oriented Languages for the Macintosh", *Byte* 16(8), pp. 177-185.

23. Stefik, M. and D. Bobrow (Winter 1986) "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, 6(4), pp. 40-62.

24. Texas Instruments (1986) An Experimental Version of SCOOPS (Source Code), Dallas, Texas: Texas Instruments.

25. Texas Instruments (1987a) *PC Scheme User's Guide*, Dallas, Texas: Texas Instruments.

26. Texas Instruments (1987b) *TI Scheme Language Reference Manual*, Dallas, Texas: Texas Instruments.

27. The Whitewater Group (1987) *Actor Language Manual*, Evanston, Illinois: The Whitewater Group.

28. The Xerox Learning Research Group (August 1981) "The Smalltalk-80 System", *Byte*, 6(8), pp. 36-48.

APPENDIX A

A DESCRIPTION OF GALAHAD'S CLASS AND INSTANCE VECTORS

The purpose of this appendix is to provide the reader a brief analysis of Galahad's class and instance vectors. The information conveyed in this section is meant to supplement the material presented in Chapter 4, The Galahad System. This appendix is not to be considered complete unless accompanied by the main text of this paper.

Galahad Class Vector

The 30 slot vector we used to implement Galahad classes is described below. Items appearing entirely in capital letters should be interpreted as literals. Ellipses indicate that a particular list or atom is repeated as many times as necessary.

Vector Position 0

Label. Galahad Class Identifier

Format. GALAHAD-CLASS

Description. This position is a label indicating the vector represents a Galahad class.

Vector Position 1

Label. Class Name

Format. Classname

Description. The Class Name position identifies the name of the Galahad class represented by this particular vector. The Galahad system uses this position whenever it needs to retrieve the name of the class. The Class Name slot is assigned its value during the CREATE-CLASS process.

Vector Position 2

Label. Class Instance Methods Environment Symbol

Format. Classname-INST

Description. This slot identifies the name of the class's instance method environment. The COMPILE-CLASS function of Galahad refers to this position when it needs to bind a newly created instance method environment to the symbol contained in this slot. This position is assigned its value during the CREATE-CLASS process.

Vector Position 3

Label. Kind Of

Format.
```
((Classname
      (CV Classvar Classvar . . . Classvar)
      (IV Instvar Instvar . . . Instvar))
                   ...
 (Classname
      (CV Classvar Classvar . . . Classvar)
      (IV Instvar Instvar . . . Instvar)))
```

Description. The Kind Of vector position contains the names of the class's immediate parents in the generalization hierarchy. It also contains those class variables (CV) and instance variables (IV) to be specifically included in the class's inheritance structure. The COMPILE-CLASS function of Galahad uses this slot to build the class's inheritance hierarchy. This position is assigned its value based on the KIND-OF clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-KIND-OF statement.

Vector Position 4

Label. Kinds

Format. ((Classname)(Classname) . . . (Classname))

Description. This slot contains the names of the class's immediate children in the inheritance hierarchy. The Galahad system uses this list during the COMPILE-CLASS process to propagate class variables, instance variables, class methods, and instance methods to the class's children. This position is assigned its value based on the KINDS clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-KINDS statement.

Vector Position 5

Label. Class Variables

Format.

```
((Classvar domain (value . . . value) (l-card u-card))
 (Classvar domain (value . . . value) (l-card u-card))
                           . . .
 (Classvar domain (value . . . value) (l-card u-card)))
```

Description. The Class Variables position contains the symbol name, variable domain, default value list, and cardinality quadruplet for each locally defined class variable. The Galahad system uses the values in this position as a kernel for building the All Class Variables slot (vector position 6). This position is assigned its value based on the CV clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-CLASS-VARIABLES statement.

Vector Position 6

Label. All Class Variables

Format.

```
((Classvar domain (value . . . value) (l-card u-card))
 (Classvar domain (value . . . value) (l-card u-card))
                           . . .
 (Classvar domain (value . . . value) (l-card u-card)))
```

Description. This slot contains the symbol name, variable domain, default value list, and cardinality quadruplets for both locally defined and all inherited class variables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position as it builds the class- variables/class-methods environment. Also, the domain and cardinality of this position are referenced each time an individual class variable receives a new value.

Vector Position 7

Label. Instance Variables

Format.
```
((Instvar domain (value . . . value) (l-card u-card))
 (Instvar domain (value . . . value) (l-card u-card))
                  . . .
 (Instvar domain (value . . . value) (l-card u-card)))
```

Description. The Instance Variables position contains the symbol name, variable domain, default value list, and cardinality quadruplet for each locally defined instance variable. The Galahad system uses the instance variables listed in this position as a kernel for building the All Instance Variables Inheritance Structure slot (vector position 8). Also, the domain and cardinality of this position are referenced each time an individual instance variable receives a new value. This slot is assigned its value based on the IV clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-INSTANCE-VARIABLES statement.

Vector Position 8

Label. All Instance Variables Inheritance Structure

Format.
```
((Instvar . Classname)(Instvar . Classname)
 (Instvar . Classname)(Instvar . Classname)
                  . . .
 (Instvar . Classname)(Instvar . Classname))
```

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited instance variables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 9

Label. Class Constraints

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
         . . .
     (domain? . . . domain? <#PROCEDURE>)))
     . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
         . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. The Class Constraints slot contains the method-name/procedure-object and the dispatch-table/table pairs for locally defined class constraint-methods. These methods have been specified by the user to be procedures to enforce user-defined constraints in a Galahad application. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function uses this slot as a kernel for building the All Class Constraints position (vector position 10). A new constraint-method is added to this slot each time the user calls CREATE-CLASS-CONSTRAINT. Conversely, constraint-methods can be removed from this position by DELETE-CLASS-CONSTRAINT as well.

Vector Position 10

Label. All Class Constraints

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
         . . .
     (domain? . . . domain? <#PROCEDURE>)))
     . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
         . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. This slot contains the method-name/procedure-object and the dispatch-table/table pairs for both locally defined and all inherited class constraint-methods. These methods have been specified by the user to be procedures to enforce user-defined class level constraints in a Galahad application. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position as it builds the class's class-variables/class-methods environment.

Vector Position 11

Label. Instance Constraints

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                . . .
     (domain? . . . domain? <#PROCEDURE>)))
                . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. The Instance Constraints slot contains the method-name/procedure-object and the dispatch-table/table pairs for locally defined instance constraint-methods. These methods have been specified by the user to be procedures to enforce user-defined constraints in a Galahad application. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function uses this slot as a kernel for building the All Instance Constraints position (vector position 12). A new constraint-method is added to this slot each time the user calls CREATE-INSTANCE-CONSTRAINT. Conversely, constraint-methods can be removed from this position by DELETE-INSTANCE-CONSTRAINT as well.

Vector Position 12

Label. All Instance Constraints

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
              . . .
     (domain? . . . domain? <#PROCEDURE>)))
              . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
              . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. This slot contains the method-name/procedure-object and the dispatch-table/table pairs for both locally defined and all inherited instance constraint-methods. These methods have been specified by the user to be procedures to enforce user-defined instance level constraints in a Galahad application. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position as it builds the class's instance-methods environment.

Vector Position 13

Label. Class Methods

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
              . . .
     (domain? . . . domain? <#PROCEDURE>)))
              . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
              . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. The Class Methods slot contains the method-name/procedure-object and the dispatch-table/table pairs for locally defined class methods. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function uses this slot as a kernel for building the All Class Methods position (vector position 14). A new method is added to this slot each time the user calls CREATE-CLASS-METHOD. Conversely, class methods can be removed from this position by DELETE-CLASS-METHOD as well.

Vector Position 14

Label. All Class Methods

Format.

```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
     ((domain? . . . domain? <#PROCEDURE>)
      (domain? . . . domain? <#PROCEDURE>)
                . . .
      (domain? . . . domain? <#PROCEDURE>)))
                . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
     ((domain? . . . domain? <#PROCEDURE>)
      (domain? . . . domain? <#PROCEDURE>)
                . . .
      (domain? . . . domain? <#PROCEDURE>)))
```

Description. This slot contains the method-name/procedure-object and the dispatch-table/table pairs for both locally defined and all inherited class methods. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position as it builds the class's class-variables/class-methods environment.

Vector Position 15

Label. Instance Methods

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                 . . .
     (domain? . . . domain? <#PROCEDURE>)))
                 . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                 . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. The Instance Methods slot contains the method-name/procedure-object and the dispatch-table/table pairs for locally defined instance methods. The reader is referred to the section on method specialization for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function uses this slot as a kernel for building the All Instance Methods position (vector position 15). A new method is added to this slot each time the user calls CREATE-INSTANCE-METHOD. Conversely, methods can be removed from this position by DELETE-INSTANCE-METHOD as well.

Vector Position 16

Label. All Instance Methods

Format.
```
((Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                 . . .
     (domain? . . . domain? <#PROCEDURE>)))
                 . . .
 (Methodname <#PROCEDURE>)
 (Methodname-DISPATCH-TABLE
    ((domain? . . . domain? <#PROCEDURE>)
     (domain? . . . domain? <#PROCEDURE>)
                 . . .
     (domain? . . . domain? <#PROCEDURE>)))
```

Description. This slot contains the method-name/procedure-object and the dispatch-table/table pairs for both locally defined and all inherited instance methods. The reader is referred to the section on method specializ ιtion for details on the relationship between methods and method dispatch tables. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position as it builds the class's instance-methods environment.

Vector Position 17

Label. Instance List

Format. (Instance-name Instance-name
. . .
Instance-name)

Description. The Instance List contains the names of instances defined for the class. It is used by Galahad class membership testing primitives to determine whether or not an instance, passed as a parameter, is a valid instance of the class. A new instance is added the class's Instance List slot each time there is a call to the %ADD-CLASS-TO-INSTANCE% Galahad Statement..

Vector Position 18

Label. Part Of

Format. ((Domain (l-card u-card))
...
(Domain (l-card u-card)))

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Part Of instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Part Of Inheritance Structure slot (vector position 19). Also, the cardinality of the domains in this position is referenced each time the Part Of aggregation becomes updated. This slot is assigned its value based on the IA/PART-OF clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-PART-OF Galahad statement.

Vector Position 19

Label. All Part Of Inheritance Structure

Format.

```
((Domain . Classname)(Domain . Classname)
 (Domain . Classname)(Domain . Classname)
                    . . .
 (Domain . Classname)(Domain . Classname))
```

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Part Of instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 20

Label. Parts

Format.

```
((Domain (l-card u-card))
          ...
 (Domain (l-card u-card)))
```

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Parts instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Parts Inheritance Structure slot (vector position 21). Also, the cardinality of the domains in this position is referenced each time the Parts aggregation becomes updated. This slot is assigned its value based on the IA/PARTS clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-PARTS Galahad statement.

Vector Position 21

Label. All Parts Inheritance Structure

Format.

```
((Domain . Classname)(Domain . Classname)
 (Domain . Classname)(Domain . Classname)
                    . . .
 (Domain . Classname)(Domain . Classname))
```

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Parts instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 22

Label. Element Of

Format. ((Domain (l-card u-card))

...

(Domain (l-card u-card)))

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Element Of instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Element Of Inheritance Structure slot (vector position 23). Also, the cardinality of the domains in this position is referenced each time the Element Of aggregation becomes updated. This slot is assigned its value based on the IA/ELEMENT-OF clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-ELEMENT-OF Galahad statement.

Vector Position 23

Label. All Element Of Inheritance Structure

Format.

((Domain . Classname)(Domain . Classname)
(Domain . Classname)(Domain . Classname)

. . .

(Domain . Classname)(Domain . Classname))

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Element Of instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 24

Label. Elements

Format. ((Domain (l-card u-card))

...

(Domain (l-card u-card)))

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Elements instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Elements Inheritance Structure slot (vector position 25). Also, the cardinality of the domains in this position is referenced each time the Elements aggregation becomes updated. This slot is assigned its value based on the IA/ELEMENTS clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-ELEMENTS Galahad statement.

Vector Position 25

Label. All Elements Inheritance Structure

Format.

((Domain . Classname)(Domain . Classname)
(Domain . Classname)(Domain . Classname)

. . .

(Domain . Classname)(Domain . Classname))

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Elements instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 26

Label. Member Of

Format. ((Domain (l-card u-card))

...

(Domain (l-card u-card)))

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Member Of instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Member Of Inheritance Structure slot (vector position 27). Also, the cardinality of the domains in this position is referenced each time the Member Of aggregation becomes updated. This slot is assigned its value based on the

IA/MEMBER-OF clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-MEMBER-OF Galahad statement.

Vector Position 27

Label. All Member Of Inheritance Structure

Format.

```
((Domain . Classname)(Domain . Classname)
 (Domain . Classname)(Domain . Classname)
                    . . .
 (Domain . Classname)(Domain . Classname))
```

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Member Of instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Vector Position 28

Label. Members

Format.

```
((Domain (l-card u-card))
           ...
 (Domain (l-card u-card)))
```

Description. This position contains the domain/cardinality pairs for each locally defined domain of the Members instance aggregation. The COMPILE-CLASS function of Galahad uses the domains listed in this position as a kernel for building the All Members Inheritance Structure slot (vector position 29). Also, the cardinality of the domains in this position is referenced each time the Members aggregation becomes updated. This slot is assigned its value based on the IA/MEMBERS clause passed to CREATE-CLASS. It also can be assigned a value from direct use of the ADD-MEMBERS Galahad statement.

Vector Position 29

Label. All Members Inheritance Structure

Format.

```
((Domain . Classname)(Domain . Classname)
 (Domain . Classname)(Domain . Classname)
                  . . .
 (Domain . Classname)(Domain . Classname))
```

Description. This position contains the symbol/defining-class pairs for both locally defined and all inherited domains for the Members instance aggregation. The COMPILE-CLASS function of Galahad both uses and assigns values to this slot. COMPILE-CLASS first assigns values to this position based on the class's inheritance hierarchy. It then uses this position to build the instance environments needed for instance method evaluation.

Galahad Instance Vector

The 9 slot vector described below is the main data structure we used to model Galahad instances. As with our description of the class vector, items appearing entirely in capital letters should be interpreted as literals. Ellipses indicate that a particular list or atom can be repeated as many times as necessary.

Vector Position 0

Label. Instance Name

Format. Instance-name

Description. The Instance Name position identifies the name of the Galahad instance represented by this particular vector. The Galahad system uses this position whenever it needs to retrieve the name of the instance. The Instance Name slot is assigned its value during the %CREATE-INSTANCE% process.

Vector Position 1

Label. All Instance Variables

Format.

```
((Classname (Instvar Value) ... (Instvar Value)
 (Classname (Instvar Value) ... (Instvar Value))
                  . . .
 (Classname (Instvar Value) ... (Instvar Value)))
```

Description. This slot contains an ASSOC list of symbol/value pairs for all instance variables represented by the instance object. The CAR of each ASSOC list component is the name of the class locally defining the corresponding instance variables. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components, complete with symbol/value pairs for each call to %ADD-CLASS-TO-INSTANCE%. The value portion of each symbol/value pair is updated whenever the corresponding instance variable is assigned a new value.

Vector Position 2

Label. All Part Of

Format.

```
((Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                  . . .
    (Domain Instance-Name . . . Instance-Name))
 (Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                  . . .
    (Domain Instance-Name . . .Instance-Name)))
```

Description. The All Part Of position contains a nested ASSOC list of instance names forming the Part Of aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Part Of specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name . . . Instance-Name* portion of each component is updated whenever the Part Of aggregation receives a new value for the instance.

Vector Position 3

Label. All Parts

Format.

```
((Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                   . . .
    (Domain Instance-Name . . . Instance-Name))
 (Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                   . . .
    (Domain Instance-Name . . .Instance-Name)))
```

Description. This slot contains a nested ASSOC list of instance names forming the Parts aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Parts specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name . . . Instance-Name* portion of each component is updated whenever the Parts aggregation receives a new value for the instance.

Vector Position 4

Label. All Element Of

Format.

```
((Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                   . . .
    (Domain Instance-Name . . . Instance-Name))
 (Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                   . . .
    (Domain Instance-Name . . .Instance-Name)))
```

Description. The All Element Of position contains a nested ASSOC list of instance names forming the Element Of aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Element Of specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name . . . Instance-Name* portion of each component is updated whenever the Element Of aggregation receives a new value for the instance.

Vector Position 5

Label. All Elements

Format.

```
((Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                      . . .
    (Domain Instance-Name . . . Instance-Name))
 (Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                      . . .
    (Domain Instance-Name . . .Instance-Name)))
```

Description. This slot contains a nested ASSOC list of instance names forming the Elements aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Elements specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name . . . Instance-Name* portion of each component is updated whenever the Elements aggregation receives a new value for the instance.

Vector Position 6

Label. All Member Of

Format.

```
((Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                      . . .
    (Domain Instance-Name . . . Instance-Name))
 (Classname
    (Domain Instance-Name . . . Instance-Name)
    (Domain Instance-Name . . . Instance-Name)
                      . . .
    (Domain Instance-Name . . .Instance-Name)))
```

Description. The All Member Of position contains a nested ASSOC list of instance names forming the Member Of aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Member Of specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name* . . . *Instance-Name* portion of each component is updated whenever the Member Of aggregation receives a new value for the instance.

Vector Position 7

Label. All Members

Format.

```
((Classname
     (Domain Instance-Name . . . Instance-Name)
     (Domain Instance-Name . . . Instance-Name)
                    . . .
     (Domain Instance-Name . . . Instance-Name))
 (Classname
     (Domain Instance-Name . . . Instance-Name)
     (Domain Instance-Name . . . Instance-Name)
                    . . .
     (Domain Instance-Name . . .Instance-Name)))
```

Description. This slot contains a nested ASSOC list of instance names forming the Members aggregation for this particular instance. The CAR of each nested ASSOC list component is the domain (defining class name) of the corresponding listed instances. The head of each main component is the name of the class whose Members specification locally defines the nested domains. The SEND mechanism in Galahad uses this slot to build the temporary instance method evaluation environment. This slot is assigned new class components for each call to %ADD-CLASS-TO-INSTANCE%. The *Instance-Name* . . . *Instance-Name* portion of each component is updated whenever the Members aggregation receives a new value for the instance.

Vector Position 8

Label. Instance Of

Format. (Classname . . . Classname)

Description. The Instance Of position contains the names of the defining classes for this particular instance. The Galahad system uses this slot to determine the classes in which to evaluate the object's instance methods. This slot is updated for each call to %ADD-CLASS-TO-INSTANCE%.

APPENDIX B

GALAHAD USER'S GUIDE

Galahad is an object-oriented conceptual modeling language designed to run on DOS-based micro computers equipped with a PC Scheme compiler. The purpose of this appendix is to provide you with an overview of how to use the Galahad system to build modeling applications. Since our emphasis will be more procedural than theoretical, we assume you have some degree of familiarity with object-oriented concepts. Additionally, we presume you have read at least the first and second chapters of this thesis.

Appendix B is divided into two main sections. The first section, Galahad Overview, provides a top level description of how to use the system. Topics include loading and starting Galahad, designing a Galahad application, entering the design into the system, and controlling Galahad's operation. The second section, Galahad Statements, describes the form and function of each Galahad statement.

Galahad Overview

As stated above, the Galahad system is designed to run under PC Scheme on any DOS or DOS-compatible machine. We recommend, however, that you use at least a 10 MHz IBM PC/AT with one megabyte of memory. Otherwise, you will not be able to model an application of any appreciable size. Also, the response time for slower machines is prohibitive.

Other than the syntax of the language, the system is fairly simple to operate. Because of its simple design, however, it is extremely easy to fall into the Scheme Inspector. The current version of Galahad has no trapping mechanism for catching Scheme errors cause by invalid user input. Should you accidently enter the inspector, exit by pressing <Cntl-Q> and type *(start-galahad)*. This should return you to the Galahad prompt.

Loading and Starting Galahad

To load Galahad, first place the Galahad system disk in the A: drive. If you have not already done so, start the PC Scheme compiler. After Scheme is loaded, type *(load "a:galahad.s")*. The system will automatically start and display the Galahad prompt after all system files have been loaded. To assist you in monitoring the progress of the load, Galahad displays the names of each individual file as they are being read into the system.

Designing a Galahad Application

While it is not the intent of this paper to double as a tutorial on object-oriented design, we do include a few general guidelines for you to follow.

First, identify all classes, subclasses, and simple classes to be included in the system. While Galahad does provide primitives for altering classes and instances after they have been entered into the system, implementation will proceed more smoothly if you have thought through the design first.

Second, identify all required instances to be modeled. Also include with these instances the names of their defining class(es). Again, the time spent in a thorough design will result in a greater time savings during implementation.

Finally, identify all inter-class and inter-instance connections. Be sure to include all appropriate reverse connections. An example of a forward/reverse connection is the Member-Of/Members relationship.

Entering a Design into the System

Most Galahad applications are entered into the system via the EDWIN editor. If so desired, you can also type any Galahad command or message directly at the Galahad prompt. Regardless of the method you choose, it is important to realize that portions of most Galahad commands and messages are treated as evaluated LISP expressions. This provides you with the capability for nesting Galahad messages or mixing messages with standard LISP commands. For example, the message *(send (send uss-missouri get-captain) get-rank)* returns the rank of the officer in charge of the USS Missouri.

Figure B-1 shows the proper overall format for a Galahad application. Notice the application is divided into separate sections for Galahad classes and instances. This is due to the system's assumption that all classes have been defined before instances can be created. Also, notice the first statement in an application should always be *(delay-compile)*, and the last statement should always be *(compile)*.

The DELAY-COMPILE function temporarily suspends domain constraint enforcement. This temporary suspension allows you to model mutual interclass dependencies. For example, assume the class CONVOY has a Members

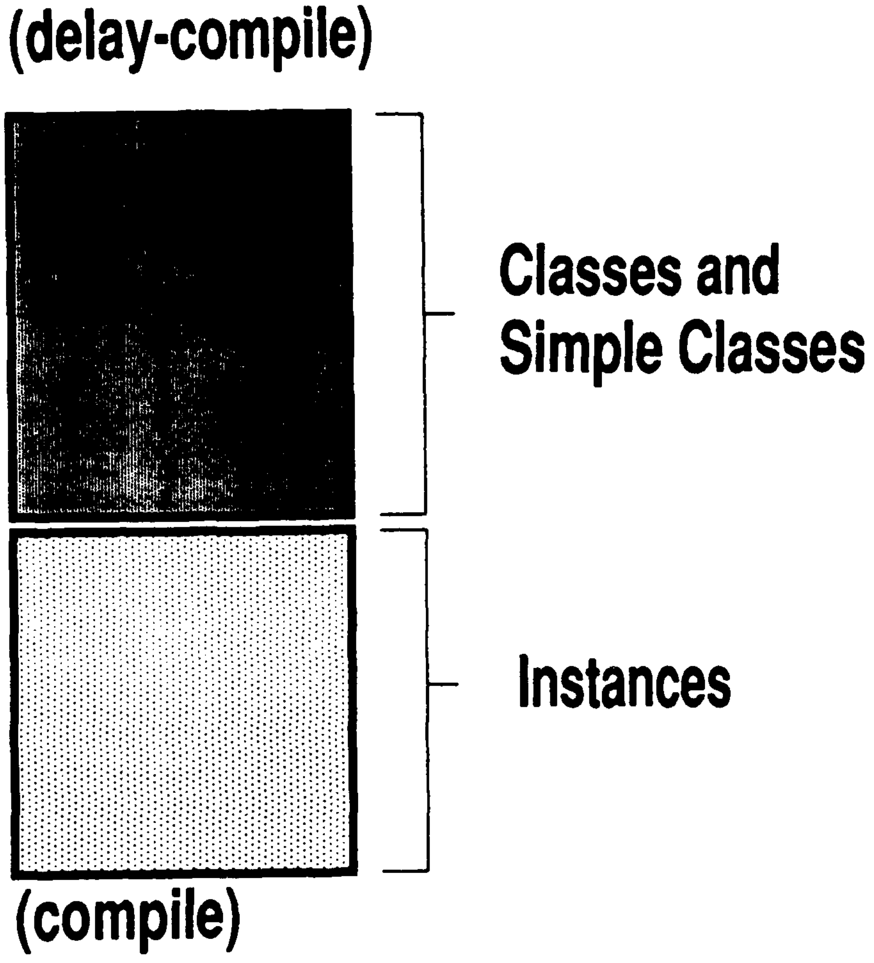


Figure B-1. Proper Overall Format for a Galahad Application instance aggregation connection to the class SHIP. Conversely, the class SHIP has a Member-Of connection to CONVOY. DELAY-COMPILE enables you to enter both of these classes into the system before Galahad checks for this mutual interclass dependency. The COMPILE command informs the system that all user-supplied input is complete and the system should examine the application for domain constraint violations.

Controlling Galahad's Operation

Thus far, we have already introduced the START-GALAHAD, DELAY-COMPILE, and COMPILE control functions. Two other functions that are useful for controlling the system are CHECK-CLASSES-FOR-STRUCTURAL-INTEGRITY and RESET-GALAHAD.

CHECK-CLASSES-FOR-STRUCTURAL-INTEGRITY allows you to examine the class structure of your model to ensure all interclass connections requiring "reverse links" have been completed. For example, this function would detect whether a Members link between CONVOY and SHIP is lacking the necessary Member-Of connection between SHIP and CONVOY.

RESET-GALAHAD enables you to remove a Galahad application without having to completely exit and reload the system. This feature is useful whenever you want to remove one application and read in another.

Galahad Statements

This section describes all Galahad commands and methods that comprise the current version of the system. For each item, we provide both the syntax and an explanation for the statement's use. We also include examples to illustrate how each item can be used in a Galahad application.

In our presentation of a statement's format, we write key words in capital letters. Optional parameters are surrounded by braces ([]). Quoted items identify evaluated S-expressions or symbols.

Many of the statements described below have the optional key word NOCOMPILE. This key word tells Galahad not to compile the class after the class has been modified. NOCOMPILE is used only if additional modifications are to be to made to the class, and the user desires to save processing time by waiting until all changes have been made before allowing the class to be compiled.

ADD-CLASS-VARIABLES

Format:

```
(SEND 'METACLASS 'ADD-CLASS-VARIABLES
      'classname
      '((classvar domain (value . . . value) (l-card u-card))
       (classvar domain (value . . . value) (l-card u-card))
                              . . .
       (classvar domain (value . . . value) (l-card u-card)))
      ['NOCOMPILE])
```

Description:

ADD-CLASS-VARIABLES is responsible for adding class variables to a previously created Galahad class.

Each class variable is defined by a list of length four. ***Classvar*** is the name of the variable to be added to the class. ***Domain*** identifies the class of legal values the variable can assume. The ***(Value . . . Value)*** field corresponds to the variable's default value(s). ***(l-card u-card)*** specifies the variable's cardinality.

Example:

```
(send 'metaclass 'add-class-variables
      'war-ship
      '((annual-budget dollars ($5,000,000,000) (1 1)))
```

In this example, we are adding the class variable Annual-Budget to the class WAR-SHIP. The domain for Annual-Budget is DOLLARS, and the default value is five billion dollars. This particular class variable is mandatory due to the cardinality of (1 1).

ADD-CLASS-TO-INSTANCE

Format:

```
(SEND 'classname '%ADD-CLASS-TO-INSTANCE%
    'new-classname
    'instance-name
    ['(
        [(PART-OF
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))]
        [(PARTS
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))]
        [(ELEMENT-OF
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))]
        [(ELEMENTS
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))]
        [(MEMBER-OF
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))]
        [(MEMBERS
            (instance-name [instance-class])
            (instance-name [instance-class])
                  . . .
            (instance-name [instance-class]))] )]
    ['((instvar (values)) . . . (instvar (values)))]
```

Description:

ADD-CLASS-TO-INSTANCE is responsible for adding another defining class to a previously created instance. This method is designed specifically to support the instance lattice inheritance concept.

Classname is the name of the class responsible for originally creating the instance. This class is occasionally referred to as the "owning" class. ***New-classname*** identifies the name of the defining class to be added to ***instance-name***. For each instance aggregation specification, ***instance-name*** is the name of the instance which will automatically create the reverse connection. ***Instance-class*** is the defining class of ***instance-name*** and is used to determine the class of the reverse instance aggregation connection. ***Instance-class*** should be used only if ***instance-name*** belongs to more than one class. ***Instvar*** and ***values*** are used to provide initial values to the instance after the class has been added.

Example:

```
(send 'student '%add-class-to-instance%
      'professor
      'dpt
      '((member-of (CU-faculty-association)))
      '((address ("Room 231 Business Bldg"))
        (research-interest ("conceptual-modeling"))))
```

In this example, we are adding the PROFESSOR class to the instance DPT. The original class defining DPT is STUDENT. We also show an initial Member-of instance aggregation and we initialize two variables: Address and Research-Interest.

ADD-ELEMENT-OF (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-ELEMENT-OF
      classname
      ((domain (l-card u-card)) . . . (domain (l-card u-card)))
      ['NOCOMPILE])
```

Description:

The METACLASS method ADD-ELEMENT-OF is responsible for adding the Element-Of instance aggregation specification to a previously created Galahad class.

Each Element-Of specification is defined by a list of length two. The name of the class responsible for originating the reverse Elements specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-element-of
      'officer
      '((assignment (0 1))))

(send 'metaclass 'add-elements
      'assignment
      '((officer (1 1))))
```

In this example, we are building an Element-Of link between OFFICER and ASSIGNMENT. We also show the reverse Elements connection from ASSIGNMENT to OFFICER. The cardinality for this particular example indicates that all ASSIGNMENTs must have OFFICERs, however, an OFFICER may or may not be an Element-Of an assignment.

ADD-ELEMENT-OF (instance method)

Format:

```
(SEND 'instance-name 'ADD-ELEMENT-OF
      'element-of-instance-name
      '[element-of-instance-name-defining-class])
```

Description:

The instance method ADD-ELEMENT-OF is responsible for adding the Element-Of/Elements instance aggregation connection to a previously created Galahad instance. Unlike ADD-ELEMENT-OF for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-ELEMENTS.

The name of the instance originating the reverse Elements link is specified by ***element-of-instance-name***. The optional ***element-of-instance-name-defining-class*** enables the user to specify the defining class of ***element-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('officer) '%create-instance% 'officer-1)
```

```
(send ('assignment) '%create-instance% 'assignment-1)

(send 'assignment '%add-class-to-instance% 'any-other-class
      'assignment-1)

(send 'officer-1 'add-element-of
      'assignment-1 'assignment)
```

In this example, we build an Element-of/Elements connection between officer-1 and assignment-1. Since assignment-1 is an instance of both ASSIGNMENT and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Elements connection.

ADD-ELEMENTS (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-ELEMENTS
      classname
      ((domain (l-card u-card)) . . . (domain (l-card u-card)))
      ['NOCOMPILE])
```

Description:

The METACLASS method ADD-ELEMENTS is responsible for adding the Elements instance aggregation specification to a previously created Galahad class.

Each Elements specification is defined by a list of length two. The name of the class responsible for originating the reverse Element-Of specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-elements
      'assignment
      '((officer (1 1))))

(send 'metaclass 'add-element-of
      'officer
      '((assignment (0 1))))
```

In this example, we are building an Elements link between ASSIGNMENT and OFFICER. We also show the reverse Element-Of connection from OFFICER to ASSIGNMENT. The cardinality for this particular example indicates that all ASSIGNMENTs must have

OFFICERs, however, an OFFICER may or may not be an Element-Of an assignment.

ADD-ELEMENTS (instance method)

Format:

```
(SEND 'instance-name 'ADD-ELEMENTS
    'elements-instance-name
    '[elements-instance-name-defining-class])
```

Description:

The instance method ADD-ELEMENTS is responsible for adding the Elements/Element-Of instance aggregation connection to a previously created Galahad instance. Unlike ADD-ELEMENTS for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-ELEMENT-OF.

The name of the instance originating the reverse Element-Of link is specified by ***elements-instance-name***. The optional ***elements-instance-name-defining-class*** enables the user to specify the defining class of ***elements-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('assignment) '%create-instance% 'assignment-1)

(send ('officer) '%create-instance% 'officer-1)

(send 'officer '%add-class-to-instance% 'any-other-class
    'officer-1)

(send 'assignment-1 'add-elements
    'officer-1 'officer)
```

In this example, we build an Elements/Element-of connection between assignment-1 and officer-1. Since officer-1 is an instance of both ASSIGNMENT and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Element-Of connection.

ADD-INSTANCE-VARIABLES

Format:

```
(SEND 'METACLASS 'ADD-INSTANCE-VARIABLES
     'classname
     '((instvar domain (value . . . value) (l-card u-card))
       (instvar domain (value . . . value) (l-card u-card))
                                . . .
      (instvar domain (value . . . value) (l-card u-card)))
     ['NOCOMPILE])
```

Description:

ADD-INSTANCE-VARIABLES is responsible for adding instance variables to a previously created Galahad class.

Each instance variable is defined by a list of length four. ***Instvar*** is the name of the variable to be added to the class. ***Domain*** identifies the class of legal values the variable can assume. The ***(Value . . . Value)*** field corresponds to the variable's default value(s). ***(l-card u-card)*** specifies the variable's cardinality.

Example:

```
(send 'metaclass 'add-instance-variables
  'war-ship
  '((registry country (USA) (1 1)))
```

In this example, we are adding instance variable Registry to the class WAR-SHIP. The domain for Registry is COUNTRY, and the default value is USA. This particular class variable is mandatory due to the cardinality of (1 1).

ADD-KIND-OF

Format:

```
(SEND 'METACLASS 'ADD-KIND-OF
     'classname
     '((superclass
          [(CV classvar classvar . . . classvar)]
          [(IV instvar instvar . . . instvar)] )
                        . . .
       (superclass
          [(CV classvar classvar . . . classvar)]
          [(IV instvar instvar . . . instvar)] ) )
     ['NOCOMPILE] )
```

Description:

ADD-KIND-OF is responsible for connecting a previously created Galahad class to a new super class. The addition of the KIND-OF link will enable the newly connected class to inherit methods and instances from the super class.

Classname identifies the name of the class to receive the KIND-OF property. ***Superclass*** identifies the name of class responsible for the reverse KINDS connection.

Inheritance of variables in a lattice network is resolved by using the "left up to joins" rule; however, in those situations where there is the potential for inheritance conflict, the user can select specific instance variables by using the **IV** key word and specifying the desired ***instvars***. Class variables can be selected in the same manner by using the **CV** key word and specifying individual ***classvars***.

Example:

```
(send 'metaclass 'add-kind-of
  'submarine
  '((nuclear-powered-vehicle)
     (water-vehicle (iv size))))

(send 'metaclass 'add-kinds
     'nuclear-powered-vehicle
     '(submarine))

(send 'metaclass 'add-kinds
     'water-vehicle
     '(submarine))
```

In this example, we are specifying that NUCLEAR-POWERED-VEHICLE and WATER-VEHICLE are to be super classes of SUBMARINE. Additionally, the instance variable Size has been specifically selected to be inherited from WATER-VEHICLE. SUBMARINE inherits all other variables by following the standard "left up to joins" protocol.

ADD-KINDS

Format:

```
(SEND 'METACLASS 'ADD-KINDS
     'classname
  '(subclass subclass . . . subclass)
     ['NOCOMPILE])
```

Description:

ADD-KINDS is responsible for connecting a previously created Galahad class to a new subclass. The addition of the KINDS link will enable the newly connected subclass to inherit methods and instances from ***classname***.

Classname identifies the name of the class to receive the KINDS property. ***Subclass*** identifies the name of class responsible for the reverse KIND-OF connection.

Example:

```
(send 'metaclass 'add-kinds
      'water-vehicle
      '(submarine))

(send 'metaclass 'add-kind-of
  'submarine
  '((water-vehicle (iv size))))
```

In this example, we are specifying that WATER-VEHICLE is to be a super class of SUBMARINE.

ADD-MEMBER-OF (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-MEMBER-OF
      classname
      ((domain (l-card u-card)) . . . (domain (l-card u-card)))
      ['NOCOMPILE])
```

Description:

The METACLASS method ADD-MEMBER-OF is responsible for adding the MEMBER-Of instance aggregation specification to a previously created Galahad class.

Each MEMBER-Of specification is defined by a list of length two. The name of the class responsible for originating the reverse Members specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-member-of
      'war-ship
      '((convoy (0 1))))

(send 'metaclass 'add-members
      'convoy
      '((war-ship (2 n))))
```

In this example, we are building a Member-Of link between WAR-SHIP and CONVOY. We also show the reverse Members connection from CONVOY to WAR-SHIP. The cardinality for this particular example indicates that a WAR-SHIP can belong at most to only one CONVOY; however, a CONVOY needs at least two WAR-SHIPS.

ADD-MEMBER-OF (instance method)

Format:

```
(SEND 'instance-name 'ADD-MEMBER-OF
      'member-of-instance-name
      '[member-of-instance-name-defining-class])
```

Description:

The instance method ADD-MEMBER-OF is responsible for adding the Member-Of/Members instance aggregation connection to a previously created Galahad instance. Unlike ADD-MEMBER-OF for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-MEMBERS.

The name of the instance originating the reverse Members link is specified by ***member-of-instance-name***. The optional ***member-of-instance-name-defining-class*** enables the user to specify the defining class of ***member-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('war-ship) '%create-instance% 'war-ship-1)

(send ('convoy) '%create-instance% 'convoy-1)

(send 'convoy '%add-class-to-instance% 'any-other-class
      'convoy-1)
```

```
(send 'officer-1 'add-member-of
      'convoy-1 'convoy)
```

In this example, we build an Member-of/Members connection between war-ship-1 and convoy-1. Since convoy-1 is an instance of both CONVOY and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Members connection.

ADD-MEMBERS (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-MEMBERS
      classname
      ((domain (l-card u-card)) . . . (domain (l-card u-card)))
      ['NOCOMPILE])
```

Description:

The METACLASS method ADD-MEMBERS is responsible for adding the Members instance aggregation specification to a previously created Galahad class.

Each Members specification is defined by a list of length two. The name of the class responsible for originating the reverse member-Of specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-members
      'convoy
      '((war-ship (2 n))))

(send 'metaclass 'add-member-of
      'war-ship
      '((convoy (0 1))))
```

In this example, we are building a Members link between CONVOY and WAR-SHIP. We also show the reverse Member-of connection from WAR-SHIP to CONVOY. The cardinality for this particular example indicates that a WAR-SHIP can belong at most to only one CONVOY; however, a CONVOY needs at least two WAR-SHIPS.

ADD-MEMBERS (instance method)

Format:

```
(SEND 'instance-name 'ADD-MEMBERS
      'members-instance-name
      '[members-instance-name-defining-class])
```

Description:

The instance method ADD-MEMBERS is responsible for adding the Members/Member-Of instance aggregation connection to a previously created Galahad instance. Unlike ADD-MEMBERS for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-MEMBER-OF.

The name of the instance originating the reverse Members link is specified by ***members-instance-name***. The optional ***members-instance-name-defining-class*** enables the user to specify the defining class of ***members-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('convoy) '%create-instance% 'convoy-1)

(send ('war-ship) '%create-instance% 'war-ship-1)

(send 'war-ship '%add-class-to-instance% 'any-other-class
      'war-ship-1)

(send 'convoy-1 'add-members
      'war-ship-1 'warship)
```

In this example, we build an Members/Member-Of connection between convoy-1 and war-ship-1. Since war-ship-1 is an instance of both WAR-SHIP and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Member-Of connection.

ADD-PART-OF (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-PART-OF
     classname
     ((domain (l-card u-card)) . . . (domain (l-card u-card)))
     ['NOCOMPILE])
```

Description:

The METACLASS method ADD-PART-OF is responsible for adding the Part-Of instance aggregation specification to a previously created Galahad class.

Each Part-Of specification is defined by a list of length two. The name of the class responsible for originating the reverse Parts specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-part-of
     'missile-launcher
     '((war-ship (0 1))))

(send 'metaclass 'add-parts
     'war-ship
     '((missile-launcher (0 3))))
```

In this example, we are building a Part-Of link between MISSILE-LAUNCHER and WAR-SHIP. We also show the reverse Parts connection from WAR-SHIP to MISSILE-LAUNCHER. The cardinality for this particular example indicates that MISSILE-LAUNCHER belongs at most to one WAR-SHIP. Also a WAR-SHIP may have between zero to three MISSILE-LAUNCHERs.

ADD-PART-OF (instance method)

Format:

```
(SEND 'instance-name 'ADD-PART-OF
     'part-of-instance-name
     '[part-of-instance-name-defining-class])
```

Description:

The instance method ADD-PART-OF is responsible for adding the Part-Of/Parts instance aggregation connection to a previously created Galahad instance. Unlike ADD-PART-OF for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-PARTS.

The name of the instance originating the reverse Parts link is specified by ***part-of-instance-name***. The optional ***part-of-instance-name-defining-class*** enables the user to specify the defining class of ***part-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('missile-launcher) '%create-instance%
      'missile-launcher-1)

(send ('war-ship) '%create-instance% 'war-ship-1)

(send 'war-ship '%add-class-to-instance% 'any-other-class
      'war-ship-1)

(send 'missile-launcher-1 'add-part-of
      'war-ship-1 'war-ship)
```

In this example, we build a Part-of/Parts connection between missile-launcher-1 and war-ship-1. Since war-ship-1 is an instance of both WAR-SHIP and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Parts connection.

ADD-PARTS (METACLASS METHOD)

Format:

```
(SEND 'METACLASS 'ADD-PARTS
      classname
      ((domain (l-card u-card)) . . . (domain (l-card u-card)))
      ['NOCOMPILE])
```

Description:

The METACLASS method ADD-PARTS is responsible for adding the Parts instance aggregation specification to a previously created Galahad class.

Each Parts specification is defined by a list of length two. The name of the class responsible for originating the reverse Part-Of specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

Example:

```
(send 'metaclass 'add-parts
      'war-ship
      '((missile-launcher (0 3))))

(send 'metaclass 'add-part-of
      'missile-launcher
      '((war-ship (0 1))))
```

In this example, we are building a Parts link between WAR-SHIP and MISSILE-LAUNCHER. We also show the reverse Part-of connection from MISSILE-LAUNCHER to WAR-SHIP. The cardinality for this particular example indicates that MISSILE-LAUNCHER belongs at most to one WAR-SHIP. Also a WAR-SHIP may have between zero to three MISSILE-LAUNCHERs.

ADD-PARTS (instance method)

Format:

```
(SEND 'instance-name 'ADD-PARTS
      'part-of-instance-name
      '[part-of-instance-name-defining-class])
```

Description:

The instance method ADD-PARTS is responsible for adding the Parts/Part-Of instance aggregation connection to a previously created Galahad instance. Unlike ADD-PARTS for METACLASS, this instance method automatically establishes the reverse link. Consequently, the behavior of this method is identical to that of the instance method ADD-PART-OF.

The name of the instance originating the reverse Part-Of link is specified by ***parts-instance-name***. The optional ***parts-instance-name-defining-class*** enables the user to specify the defining class of ***parts-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('war-ship) '%create-instance% 'war-ship-1)
```

```
(send ('missile-launcher) '%create-instance%
      'missile-launcher-1)

(send 'missile-launcher '%add-class-to-instance% 'any-other-class
      'missile-launcher-1)

(send 'war-ship-1 'add-parts
      'missile-launcher-1 'missile-launcher)
```

In this example, we build a Parts/Part-of connection between war-ship-1 and missile-launcher-1. Since missile-launcher-1 is an instance of both MISSILE-LAUNCHER and ANY-OTHER-CLASS, we must specify the defining class to be used in completing the reverse Part-Of connection.

CHECK-CLASSES-FOR-STRUCTURAL-INTEGRITY

Format:

```
(CHECK-CLASSES-FOR-STRUCTURAL-INTEGRITY)
```

Description:

This Galahad control function examines the current application in the system and searches for incomplete interclass connections. If any incomplete connections are found, this function prints a message informing the user of the problem.

Example:

```
(send 'mctaclass 'create-class 'submarine
      (kind-of ((war-ship))))

(check-classes-for-structural-integrity)
```

In this example, CHECK-CLASSES-FOR-STRUCTURAL-INTEGRITY would print an error message indicating a mandatory KINDS connection was missing between WAR-SHIP and SUBMARINE.

COMPILE

Format:

```
(COMPILE)
```

Description:

This Galahad control function works in conjunction with the DELAY-COMPILE command. COMPILE turns off DELAY-COMPILE mode in the system and starts checking the user model currently in the system for domain constraint violations. If any violations occur, they are immediately reported to the user.

Example:

```
(delay-compile)

(send 'metaclass 'create-class 'submarine
    (kind-of ((war-ship)))
    (iv (maximum-depth feet (#!unassigned) (1 1))))

(compile)
```

In this example, assuming no other classes exist in the system, Galahad would return an error message indicating that FEET is an unknown class/simple-class. The error would be returned only after the COMPILE command has been called.

COMPILE-CLASS

Format:

```
(SEND 'METACLASS 'COMPILE-CLASS
    'classname
    'compiler-option)

Where compiler-option = {CLASS-VARIABLES
                         CLASS-METHODS
                         INSTANCE-VARIABLES
                         INSTANCE-METHODS
                         PART-OF
                         PARTS
                         ELEMENT-OF
                         ELEMENTS
                         MEMBER-OF
                         MEMBERS
                         ALL}
```

Description:

COMPILE-CLASS is responsible for completing the specification of ***classname*** by causing the class to inherit variables and methods from its superclasses. As a general rule, COMPILE-CLASS should never be called directly from a user's application because Galahad automatically completes the inheritance structure whenever a class is altered. The only time the user should call COMPILE-CLASS is if he/she has specified the NOCOMPILE option on a method altering the structure of a class.

Compiler-option contains a key word passed to COMPILE-CLASS that specifies which class links should be processed. This is used strictly for the purpose of system efficiency.

Example:

```
(send 'metaclass 'compile-class 'submarine
      'instance-variables)

(send 'metaclass 'compile-class 'submarine
      'all)
```

In this example, we are forcing the class SUBMARINE to complete its inheritance structure. In the first example, we are only completing the inheritance of SUBMARINE's instance variables. In the second example, all variable, method, and instance aggregation properties are processed.

CREATE-CLASS

Format:

```
(SEND 'METACLASS 'CREATE-CLASS
     'classname
     ['(KIND-OF
          (superclass
               [(CV classvar classvar . . . classvar)]
               [(IV instvar instvar . . . instvar))]
                         . . .
          (superclass
               [(CV classvar classvar . . . classvar)]
               [(IV instvar instvar . . . instvar))]
     ['(KINDS subclass subclass . . . subclass)]
     ['(CV (classvar domain (value . . . value) (l-card u-card))
           (classvar domain (value . . . value) (l-card u-card))
                         . . .
           (classvar domain (value . . . value) (l-card u-card))
     ['(IV (instvar domain (value . . . value) (l-card u-card))
           (instvar domain (value . . . value) (l-card u-card))
                         . . .
           (instvar domain (value . . . value) (l-card u-card))
     ['(IA
          [(PART-OF
               (domain (l-card u-card)) . . . (domain (l-card u-card))]
          [(PARTS
               (domain (l-card u-card)) . . . (domain (l-card u-card))]
          [(ELEMENT-OF
               (domain (l-card u-card)) . . . (domain (l-card u-card))]
          [(ELEMENTS
               (domain (l-card u-card)) . . . (domain (l-card u-card))]
          [(MEMBER-OF
               (domain (l-card u-card)) . . . (domain (l-card u-card))]
          [(MEMBERS
               (domain (l-card u-card)) . . . (domain (l-card u-card))] )]
     ['(ABSTRACT)] )
```

Description:

CREATE-CLASS is responsible for building Galahad classes and assigning to these classes variables and instance aggregation specifications.

Classname identifies the name of the class to be created.

The **KIND-OF** clause identifies the individual ***superclasses*** from which **classname** is to inherit all inheritable variables and methods. Inheritance of variables in a lattice network is resolved by using the "left up to joins" rule; however, in those situations where there is the potential for inheritance conflict, the user can select specific instance variables by using the **IV** key word and specifying the desired ***instvars***. Class variables can be selected in the same manner by using the **CV** key word and specifying individual ***classvars***.

The **KINDS** clause specifies the individual ***subclasses*** which are to be connected to ***classname*** in the inheritance hierarchy.

CV and **IV** identify the individual class and instance variables to be associated with ***classname***. ***Classvar*** is the name of each class variable. ***Instvar*** is the name of each instance variable. ***Domain*** identifies the class of legal values the variable can assume. The ***(Value . . . Value)*** field corresponds to the variable's default value(s). ***(l-card u-card)*** specifies the variable's cardinality.

The **IA** clause specifies the instance aggregation specifications for classname. The name of the class responsible for originating the reverse instance aggregation specification is identified by ***Domain***. ***(l-card u-card)*** specifies the cardinality of the connection.

The key word **ABSTRACT** identifies whether classname is to be considered an abstract class.

Example:

```
(send 'metaclass 'create-class 'submarine
      '(kind-of water-vehicle)
      '(kinds (trident attack-sub)
      '(cv (number-in-fleet number (0) (1 1)
      '(iv (max-depth number (#!unassigned)(1 1))))
      '(ia (member-of (submarine-fleet (1 1)))))
```

In this example, we are building the class SUBMARINE which is has WATER-VEHICLE as its only super class. Also, SUBMARINE has two subclasses, TRIDENT and ATTACK-SUB. There is only one class variable associated with this class, Number-In-Fleet. It is a mandatory class variable and has a default value of 0. Also, SUBMARINE has only one instance variable, Max-Depth. Max-Depth is a mandatory variable; however, there is no default value. Finally, any instance of SUBMARINE must be a Member-Of a SUBMARINE-FLEET.

CREATE-CLASS-METHOD

Format:

```
(SEND 'METACLASS 'CREATE-CLASS-METHOD
     'classname
     'method-name
     '(LAMBDA ((formal [domain]) . . . (formal [domain])
            [(OPTIONAL formal [domain] . . . [domain])])
        method-body)
     ['NOCOMPILE])
```

Description:

CREATE-CLASS-METHOD is responsible for adding a new user-defined class method to a previously created class. This function enables the user to supply behavior to a class in addition to the default behavior of the setting and retrieving of instance variables.

The name of the defining class for the new class method is specified by ***classname***. ***Method-name*** identifies the name for the new class method.

The LAMBDA list associated with the method is conventional Scheme code except for the formals specification. The formals specification is Galahad's mechanism of implementing the specialized method concept. Each ***(formal domain)*** pair identifies a parameter to be passed to the class method. ***Formal*** is the name of the parameter passed to the method. ***Domain*** identifies the class of legal values ***formal*** can assume. If ***domain*** is omitted, then ***formal*** can assume any value.

The **OPTIONAL** key word enables the user to specialize on optional parameters. If an optional parameter has been passed to a class method, then the system specializes on the parameter. Otherwise, no specialization occurs on optional arguments.

Example:

```
(send 'metaclass 'create-class-method
     'submarine
     'test-number-in-fleet
     '(lambda ((passed-self *administrator*))
          (if (< number-in-fleet 50)
             (display "Number of subs is low"))))
```

In this particular example, we are creating the class method TEST-NUMBER-IN-FLEET. This method specializes on the identity of the object sending the message. In this case, we are allowing only the system administrator to use this method. TEST-NUMBER-IN-FLEET examines the class variable Number-in-Fleet and displays a warning message if there are fewer than 50 submarines.

CREATE-INSTANCE

Format:

```
(SEND ('classname . . . 'classname) %CREATE-INSTANCE%
      'new-instance-name
      ['(defining-classname
           instvar (values) . . . instvar (values))
                  . . .
       '(defining-classname
           instvar (values) . . . instvar (values))]
```

```
['(PART-OF
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name)
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name))]
['(PARTS
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name)
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name))]
['(ELEMENT-OF
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name)
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name))]
['(ELEMENTS
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name)
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
          instance-name))]
['(MEMBER-OF
     (defining-classname
          instance-name {or}
          [(instance-name instance-class)]
               . . .
```

```
                instance-name)
            (defining-classname
                instance-name {or}
                [(instance-name instance-class)]
                    . . .
                instance-name))]
        ['(MEMBERS
            (defining-classname
                instance-name {or}
                [(instance-name instance-class)]
                    . . .
                instance-name)
            (defining-classname
                instance-name {or}
                [(instance-name instance-class)]
                    . . .
                instance-name))] )
```

Description:

CREATE-INSTANCE is responsible for building Galahad instances and assigning to these instances initial values for variables and instance aggregation connections.

(Classname . . . classname) specifies the name(s) of the defining class(es) for ***new-instance-name***. The first class listed is considered to be the "owning" class of the instance. The others are additional classes to be added to the instance. Identifying the "owning" class is important if the user intends to add another class via the ADD-CLASS-TO-INSTANCE method.

All initialization of variables and instance aggregations are processed according to their respective defining classnames. If a ***defining-classname*** is specified, it must have a corresponding ***classname*** as described above.

Instvar and ***values*** are used to provide initial values to the instance in the stated defining class.

For the instance aggregation connections, ***instance-name*** is the name of the instance which will automatically create the reverse instance aggregation connection. ***Instance-class*** is the defining class of ***instance-name*** and is used to determine the defining class of the reverse instance aggregation connection. ***Instance-class*** should be used only if ***instance-name*** belongs to more than one class.

Example:

```
(send ('student 'professor) '%create-instance%
      'dpt
      '(student
           address ("123 Elm Street")
           major ("information systems"))
      '(professor
           address ("Room 231 Business Bldg")
         research-interest ("conceptual-modeling"))
      '(member-of
           (student doctoral-student-association)
           (professor (CU-faculty-association
                              organization))))
```

In this example, we are creating the instance DPT and are assigning it two classes: STUDENT and PROFESSOR. STUDENT is considered to be the owning class for DPT.

In the STUDENT defining class, we are initializing the instance variables Address and Major. We are also specifying that DPT is a Member-Of the Doctoral-Student-Association.

In the PROFESSOR defining class, we are initializing the instance variables Address and Research-Interest. We also are specifying that DPT is a Member-Of the CU-Faculty-Association. Assuming CU-Faculty-Association is itself an instance of more than one class, we are specifying the defining class ORGANIZATION for the reverse Members connection.

CREATE-INSTANCE-METHOD

Format:

```
(SEND 'METACLASS 'CREATE-INSTANCE-METHOD
      'classname
      'method-name
      '(LAMBDA ((formal [domain]) . . . (formal [domain])
               [(OPTIONAL formal [domain] . . . [domain])])
          method-body)
      ['NOCOMPILE])
```

Description:

CREATE-INSTANCE-METHOD is responsible for adding a new user-defined instance method to a previously created class. This function enables the user to supply behavior to a class in addition to the default behavior of the setting and retrieving of instance variables.

The name of the defining class for the new class method is specified by ***classname***. ***Method-name*** identifies the name for the new instance method.

The LAMBDA list associated with the method is conventional Scheme code except for the formals specification. The formals specification is Galahad's mechanism of implementing the specialized method concept. Each ***(formal domain)*** pair identifies a parameter to be passed to the instance method. ***Formal*** is the name of the parameter passed to the method. ***Domain*** identifies the class of legal values ***formal*** can assume. If ***domain*** is omitted, then ***formal*** can assume any value.

The **OPTIONAL** key word enables the user to specialize on optional parameters. If an optional parameter has been passed to an instance method, then the system specializes on the parameter. Otherwise, no specialization occurs on optional arguments.

Example:

```
(send 'metaclass 'create-instance-method
      'submarine
      'test-current-depth
      '(lambda ((passed-self *captain*))
            (if (> current-depth maximum-allowable-depth)
               (display "exceeding maximum depth"))))
```

In this particular example, we are creating the class method TEST-CURRENT-DEPTH. This method specializes on the identity of the object sending the message. In this case, we are allowing only the submarine captain to use this method. TEST-CURRENT-DEPTH compares the instance variables Maximum-Allowable-Depth and Current-Depth. If Current-Depth is greater than Maximum-Allowable-Depth, a warning message is displayed.

CREATE-SIMPLE-CLASS

Format:

```
(SEND 'METACLASS 'CREATE-SIMPLE-CLASS
      'simple-class-name
            '(LAMBDA (formals)
                  procedure body))
```

Description:

CREATE-SIMPLE-CLASS is responsible for creating Galahad simple classes.

Simple-class-name identifies the name to be assigned to the simple class. Since simple classes are procedural specifications, the system appends a "?" to ***simple-class-name*** before binding the resulting symbol to the compiled procedure object.

The LAMBDA list can be any user-defined standard Scheme function; however, the user should write a routine that requires one parameter and returns a boolean value.

Example:

```
(send 'metaclass 'create-simple-class
     'name
     '(lambda (passed-test-value)
          (and (string? passed-test-value)
               (< (string-length passed-value)
                    25))))
```

In this example, we are creating the simple class NAME. The symbol bound to the procedure object is NAME?.

The Scheme code shown above performs a test on a passed parameter to see whether the passed argument is a string of less than 25 characters in length.

DELAY-COMPILE

Format:

```
(DELAY-COMPILE)
```

Description:

This Galahad control function works in conjunction with the COMPILE command. DELAY-COMPILE turns on DELAY-COMPILE mode, thus temporarily suspending the enforcement of domain constraint violations. COMPILE re-establishes the checking of domain constraints.

Example:

```
(delay-compile)

(send 'metaclass 'create-class 'war-ship
     (kinds submarine))

(send 'metaclass 'create-class 'submarine
     (kind-of ((war-ship))))
```

```
(compile)
```

In this example, Galahad would allow both WAR-SHIP and SUBMARINE to be created because DELAY-COMPILE mode had been turned on. Otherwise, with DELAY-COMPILE off, the system would fail in its attempt to create the first class, WAR-SHIP. This is due to the system checking for the presence of SUBMARINE, which had not yet been read into the system.

DELETE-CLASS-METHOD

Format:

```
(SEND 'METACLASS 'DELETE-CLASS-METHOD
      'classname
      'method-name
      ['NOCOMPILE])
```

Description:

DELETE-CLASS-METHOD is responsible for removing a class method from a class. This function does not distinguish among specialized methods. Consequently, one call to DELETE-CLASS-METHOD will remove all specialized methods and the generic dispatch function.

The defining class name of the method to be deleted is specified by ***classname***. ***Method-name*** identifies the name of the method.

Example:

```
(send 'metaclass 'delete-class-method
      'submarine
      'test-number-in-fleet)
```

In this particular example, we are removing from SUBMARINE the class method TEST-NUMBER-IN-FLEET. The removal includes TEST-NUMBER-IN-FLEET's generic dispatch function and all specialized methods.

DELETE-ELEMENT-OF (instance method)

Format:

```
(SEND 'instance-name 'DELETE-ELEMENT-OF
      'element-of-instance-name
      '[element-of-instance-name-defining-class])
```

Description:

The instance method DELETE-ELEMENT-OF is responsible for removing the Element-Of/Elements instance aggregation connection from a Galahad instance. Since DELETE-ELEMENT-OF removes both the forward and reverse links, its end result is identical to that achieved by DELETE-ELEMENTS.

The name of the instance originating the reverse Elements link is specified by ***element-of-instance-name***. The optional ***element-of-instance-name-defining-class*** enables the user to specify the defining class of ***element-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('officer) '%create-instance% 'officer-1
      '(element-of
           (officer assignment-1)))

(send ('assignment) '%create-instance% 'assignment-1)

(send `assignment '%add-class-to-instance% 'any-other-class
      'assignment-1)

(send 'officer-1 'delete-element-of
      'assignment-1 'assignment)
```

In this example, we remove the Element-Of/Elements connection between officer-1 and assignment-1. Since assignment-1 is an instance of both ASSIGNMENT and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Elements connection.

DELETE-ELEMENTS (instance method)

Format:

```
(SEND 'instance-name 'DELETE-ELEMENTS
      'elements-instance-name
      '[elements-instance-name-defining-class])
```

Description:

The instance method DELETE-ELEMENTS is responsible for removing the Elements/Element-Of instance aggregation connection from a Galahad instance. Since DELETE-ELEMENTS removes both the forward and reverse links, its end result is identical to that achieved by DELETE-ELEMENT-OF.

The name of the instance originating the reverse Element-Of link is specified by ***elements-instance-name***. The optional ***elements-instance-name-defining-class*** enables the user to specify the defining class of ***elements-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('assignment) '%create-instance% 'assignment-1
       '(elements
            (officer assignment-1)))

(send ('officer) '%create-instance% 'officer-1)

(send 'officer '%add-class-to-instance% 'any-other-class
      'officer-1)

(send 'assignment-1 'delete-elements
      'officer-1 'officer)
```

In this example, we remove the Elements/Element-Of connection between assignment-1 and officer-1. Since officer-1 is an instance of both OFFICER and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Element-Of connection.

DELETE-INSTANCE-METHOD

Format:

```
(SEND 'METACLASS 'DELETE-INSTANCE-METHOD
      'classname
      'method-name
      ['NOCOMPILE])
```

Description:

DELETE-INSTANCE-METHOD is responsible for removing an instance method from a class. This function does not distinguish among specialized methods. Consequently, one call to DELETE-INSTANCE-METHOD will remove all specialized methods and the generic dispatch function.

The defining class name of the method to be deleted is specified by ***classname***. ***Method-name*** identifies the name of the method.

Example:

```
(send 'metaclass 'delete-instance-method
      'submarine
      'test-current-depth)
```

In this particular example, we are removing from SUBMARINE the instance method TEST-CURRENT-DEPTH. The removal includes TEST-CURRENT-DEPTH's generic dispatch function and all specialized methods.

DELETE-MEMBER-OF (instance method)

Format:

```
(SEND 'instance-name 'DELETE-MEMBER-OF
      'member-of-instance-name
      '[member-of-instance-name-defining-class])
```

Description:

The instance method DELETE-MEMBER-OF is responsible for removing the Member-Of/Members instance aggregation connection from a Galahad instance. Since DELETE-MEMBER-OF removes both the forward and reverse links, its end result is identical to that achieved by DELETE-MEMBERS.

The name of the instance originating the reverse Members link is specified by ***member-of-instance-name***. The optional ***member-of-instance-name-defining-class*** enables the user to specify the defining class of ***member-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('war-ship) '%create-instance% 'war-ship-1
      (member-of
            (war-ship convoy-1))))

(send ('convoy) '%create-instance% 'convoy-1)

(send 'convoy '%add-class-to-instance% 'any-other-class
      'convoy-1)

(send 'war-ship-1 'delete-member-of
      'convoy-1 'convoy)
```

In this example, we remove the Member-of/Members connection between war-ship-1 and convoy-1. Since convoy-1 is an instance of both CONVOY and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Members connection.

(DELETE-MEMBERS (instance method)

Format:

```
(SEND 'instance-name 'DELETE-MEMBERS
    'members-instance-name
    '[members-instance-name-defining-class])
```

Description:

The instance method DELETE-MEMBERS is responsible for removing the Members/Member-Of instance aggregation connection from a Galahad instance. Since DELETE-MEMBERS removes both the forward and reverse links, its end result is identical to that achieved by DELETE-MEMBER-OF.

The name of the instance originating the reverse Member-Of link is specified by ***members-instance-name***. The optional ***members-instance-name-defining-class*** enables the user to specify the defining class of ***members-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('convoy) '%create-instance% 'convoy-1
    '(members
        (convoy war-ship-1))))

(send ('war-ship) '%create-instance% 'war-ship-1)

(send 'war-ship '%add-class-to-instance% 'any-other-class
    'war-ship-1)

(send 'convoy-1 'delete-members
    'war-ship-1 'warship)
```

In this example, we remove the Members/Member-Of connection between convoy-1 and war-ship-1. Since war-ship-1 is an instance of both WAR-SHIP and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Member-Of connection.

DELETE-PART-OF (instance method)

Format:

```
(SEND 'instance-name 'DELETE-PART-OF
      'part-of-instance-name
      '[part-of-instance-name-defining-class])
```

Description:

The instance method DELETE-PART-OF is responsible for removing the Part-Of/Parts instance aggregation connection from a Galahad instance. Since DELETE-PART-OF removes both the forward and reverse links, its end result is identical to that achieved by DELETE-PARTS.

The name of the instance originating the reverse Parts link is specified by ***part-of-instance-name***. The optional ***part-of-instance-name-defining-class*** enables the user to specify the defining class of ***part-of-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('missile-launcher) '%create-instance%
      'missile-launcher-1
      '(part-of (missile-launcher 'war-ship-1))))

(send ('war-ship) '%create-instance% 'war-ship-1)

(send 'war-ship '%add-class-to-instance% 'any-other-class
      'war-ship-1)

(send 'missile-launcher-1 'delete-part-of
      'war-ship-1 'war-ship)
```

In this example, we remove the Part-of/Parts connection between missile-launcher-1 and war-ship-1. Since war-ship-1 is an instance of both WAR-SHIP and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Parts connection.

DELETE-PARTS (instance method)

Format:

```
(SEND 'instance-name 'DELETE-ELEMENTS
      'elements-instance-name
      '[elements-instance-name-defining-class])
```

Description:

The instance method DELETE-ELEMENTS is responsible for removing the Elements/Element-Of instance aggregation connection from a Galahad instance. Since DELETE-ELEMENTS removes both the forward and reverse links, its end result is identical to that achieved by DELETE-ELEMENT-OF.

The name of the instance originating the reverse Element-Of link is specified by ***elements-instance-name***. The optionai ***elements-instance-name-defining-class*** enables the user to specify the defining class of ***elements-instance-name*** should that instance belong to more than one class.

Example:

```
(send ('war-ship) '%create-instance% 'war-ship-1
     '(parts (war-ship missile-launcher-1)))

(send ('missile-launcher) '%create-instance%
     'missile-launcher-1)

(send 'missile-launcher '%add-class-to-instance% 'any-other-class
     'missile-launcher-1)

(send 'war-ship-1 'delete-parts
     'missile-launcher-1 'missile-launcher)
```

In this example, we remove the Parts/Part-of connection between war-ship-1 and missile-launcher-1. Since missile-launcher-1 is an instance of both MISSILE-LAUNCHER and ANY-OTHER-CLASS, we must specify the defining class to be used in removing the reverse Part-Of connection.

GALAHAD-CLASSES

Format:

%GALAHAD-CLASSES%

Description:

%GALAHAD-CLASSES% is a symbol containing a list of all Galahad classes currently defined in the system. The user can access this symbol either directly at the Galahad prompt, or through a user-defined method.

GALAHAD-INSTANCES

Format:

%GALAHAD-INSTANCES%

Description:

%GALAHAD-INSTANCES% is a symbol containing a list of all Galahad instances currently defined in the system. The user can access this symbol either directly at the Galahad prompt, or through a user-defined method.

GALAHAD-SIMPLE-CLASSES

Format:

%GALAHAD-SIMPLE-CLASSES%

Description:

%GALAHAD-SIMPLE-CLASSES% is a symbol containing a list of all Galahad simple classes currently defined in the system. The user can access this symbol either directly at the Galahad prompt, or through a user-defined method.

GET-ALL-CLASS-CONSTRAINTS

Format:

(SEND 'classname '%GET-ALL-CLASS-CONSTRAINTS%)

Description:

GET-ALL-CLASS-CONSTRAINTS returns the contents of ***classname***'s All-Class-Constraints vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-CLASS-METHODS

Format:

(SEND 'classname '%GET-ALL-CLASS-METHODS%)

Description:

GET-ALL-CLASS-METHODS returns the contents of ***classname***'s All-Class-Methods vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, <u>A Description of Galahad's Class and Instance Vectors</u>.

GET-ALL-CLASS-VARIABLES

Format:

(SEND 'classname '%GET-ALL-CLASS-VARIABLES%)

Description:

GET-ALL-CLASS-VARIABLES returns the contents of ***classname***'s All-Class-Variables vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, <u>A Description of Galahad's Class and Instance Vectors</u>.

GET-ALL-ELEMENT-OF (class method)

Format:

(SEND 'classname '%GET-ALL-ELEMENT-OF%)

Description:

GET-ALL-ELEMENT-OF returns both locally defined and all inherited domain/cardinality pairs for the Element-Of instance aggregation.

GET-ALL-ELEMENT-OF (instance method)

Format:

(SEND 'instance-name 'GET-ALL-ELEMENT-OF)

Description:

GET-ALL-ELEMENT-OF returns the contents of ***instance-name***'s All-Element-Of vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, <u>A Description of Galahad's Class and Instance Vectors</u>.

GET-ALL-ELEMENT-OF-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-ELEMENT-OF-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-ELEMENT-OF-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Element-Of-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-ELEMENTS (class method)

Format:

(SEND 'classname '%GET-ALL-ELEMENTS%)

Description:

GET-ALL-ELEMENTS returns both locally defined and all inherited domain/cardinality pairs for the Elements instance aggregation.

GET-ALL-ELEMENTS (instance method)

Format:

(SEND 'instance-name 'GET-ALL-ELEMENTS)

Description:

GET-ALL-ELEMENTS returns the contents of ***instance-name***'s All-Elements vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-ELEMENTS-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-ELEMENTS-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-ELEMENTS-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Elements-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-INSTANCE-CONSTRAINTS

Format:

(SEND 'classname '%GET-ALL-INSTANCE-CONSTRAINTS%)

Description:

GET-ALL-INSTANCE-CONSTRAINTS returns the contents of ***classname***'s All-Instance-Constraints vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-INSTANCE-METHODS

Format:

(SEND 'classname '%GET-ALL-INSTANCE-METHODS%)

Description:

GET-ALL-INSTANCE-METHODS returns the contents of ***classname***'s All-Instance-Methods vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description oi Galahad's Class and Instance Vectors.

GET-ALL-INSTANCE-VARIABLES (class method)

Format:

(SEND 'classname '%GET-ALL-INSTANCE-VARIABLES%)

Description:

GET-ALL-INSTANCE-VARIABLES returns from the class both locally defined and all inherited instance variable specification quadruplets. A quadruplet is a list of length four that contains each instance variable's name, domain, default value(s), and cardinality.

GET-ALL-INSTANCE-VARIABLES (instance method)

Format:

(SEND 'instance-name 'GET-ALL-INSTANCE-VARIABLES)

Description:

GET-ALL-INSTANCE-VARIABLES returns the contents of ***instance-name***'s All-Instance-Variables vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-INSTANCE-VARIABLES-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-INSTANCE-VARIABLES-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-INSTANCE-VARIABLES-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Instance-Variables-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-MEMBER-OF (class method)

Format:

(SEND 'classname '%GET-ALL-MEMBER-OF%)

Description:

GET-ALL-MEMBER-OF returns both locally defined and all inherited domain/cardinality pairs for the Member-Of instance aggregation.

GET-ALL-MEMBER-OF (instance method)

Format:

(SEND 'instance-name 'GET-ALL-MEMBER-OF)

Description:

GET-ALL-MEMBER-OF returns the contents of ***instance-name***'s All-Member-Of vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-MEMBER-OF-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-MEMBER-OF-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-MEMBER-OF-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Member-Of-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-MEMBERS (class method)

Format:

(SEND 'classname '%GET-ALL-MEMBERS%)

Description:

GET-ALL-MEMBERS returns both locally defined and all inherited domain/cardinality pairs for the Members instance aggregation.

GET-ALL-MEMBERS (instance method)

Format:

(SEND 'instance-name 'GET-ALL-MEMBERS)

Description:

GET-ALL-MEMBERS returns the contents of ***instance-name***'s All-Members vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-MEMBERS-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-MEMBERS-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-MEMBERS-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Members-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-PART-OF (class method)

Format:

(SEND 'classname '%GET-ALL-PART-OF%)

Description:

GET-ALL-PART-OF returns both locally defined and all inherited domain/cardinality pairs for the Part-Of instance aggregation.

GET-ALL-PART-OF (instance method)

Format:

(SEND 'instance-name 'GET-ALL-PART-OF)

Description:

GET-ALL-PART-OF returns the contents of ***instance-name***'s All-Part-Of vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-PART-OF-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-PART-OF-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-PART-OF-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Part-Of-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-PARTS (class method)

Format:

(SEND 'classname '%GET-ALL-PARTS%)

Description:

GET-ALL-PARTS returns both locally defined and all inherited domain/cardinality pairs for the Parts instance aggregation.

GET-ALL-PARTS (instance method)

Format:

(SEND 'instance-name 'GET-ALL-PARTS)

Description:

GET-ALL-PARTS returns the contents of ***instance-name***'s All-Parts vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ALL-PARTS-INHERITANCE-STRUCTURE

Format:

(SEND 'classname '%GET-ALL-PARTS-INHERITANCE-STRUCTURE%)

Description:

GET-ALL-PARTS-INHERITANCE-STRUCTURE returns the contents of ***classname***'s All-Parts-Inheritance-Structure vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-CLASS

Format:

(SEND 'classname '%GET-CLASS%)

Description:

GET-CLASS returns the entire class vector associated with ***classname***. For details on the specific contents of this vector, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-CLASS-CONSTRAINTS

Format:

(SEND 'classname '%GET-CLASS-CONSTRAINTS%)

Description:

GET-CLASS-CONSTRAINTS returns the contents of ***classname***'s Class-Constraints vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-CLASS-METHODS

Format:

(SEND 'classname '%GET-CLASS-METHODS%)

Description:

GET-CLASS-METHODS returns the contents of ***classname***'s Class-Methods vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Calahad's Class and Instance Vectors.

GET-CLASS-NAME

Format:

(SEND 'classname '%GET-CLASS-NAME%)

Description:

GET-CLASS-NAME returns the contents of ***classname***'s Class-Name vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-CLASS-NAME-INST

Format:

(SEND 'classname '%GET-CLASS-NAME-INST%)

Description:

GET-CLASS-NAME-INST returns the contents of ***classname***'s Class-Instance-Methods-Environment-Symbol vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-CLASS-VARIABLES

Format:

(SEND 'classname '%GET-CLASS-VARIABLES%)

Description:

GET-CLASS-VARIABLES returns the contents of ***classname***'s Class-Variables vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ELEMENT-OF (class method)

Format:

(SEND 'classname '%GET-ELEMENT-OF%)

Description:

GET-ELEMENT-OF returns the contents of ***classname***'s Element-Of vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ELEMENT-OF (instance method)

Format:

(SEND 'instance-name 'GET-ELEMENT-OF)

Description:

GET-ELEMENT-OF returns the value of the Element-Of instance aggregation for ***instance-name***.

GET-ELEMENTS (class method)

Format:

(SEND 'classname '%GET-ELEMENTS%)

Description:

GET-ELEMENTS returns the contents of ***classname***'s Elements vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-ELEMENTS (instance method)

Format:

(SEND 'instance-name 'GET-ELEMENTS)

Description:

GET-ELEMENTS returns the value of the Elements instance aggregation for ***instance-name***.

GET-INSTANCE

Format:

(SEND 'instance-name 'GET-INSTANCE)

Description:

GET-INSTANCE returns the entire class vector associated with ***instance-name***. For details on the specific contents of this vector, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-CONSTRAINTS

Format:

(SEND 'classname '%GET-INSTANCE-CONSTRAINTS%)

Description:

GET-INSTANCE-CONSTRAINTS returns the contents of ***classname***'s Instance-Constraints vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-LIST

Format:

(SEND 'classname '%GET-INSTANCE-METHODS%)

Description:

GET-INSTANCE-METHODS returns the contents of ***classname***'s Instance-Methods vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-METHODS

Format:

(SEND 'classname '%GET-INSTANCE-METHODS%)

Description:

GET-INSTANCE-METHODS returns the contents of ***classname***'s Instance-Methods vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-NAME

Format:

(SEND 'instance-name 'GET-INSTANCE-NAME)

Description:

GET-INSTANCE-NAME returns the contents of ***instance-name***'s Instance-Name vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-OF

Format:

(SEND 'instance-name 'GET-INSTANCE-OF)

Description:

GET-INSTANCE-OF returns the contents of ***instance-name***'s Instance-Of vector position. For details on the specific contents of this instance vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-VARIABLES (class method)

Format:

(SEND 'classname '%GET-INSTANCE-VARIABLES%)

Description:

GET-INSTANCE-VARIABLES returns the contents of ***classname***'s Instance-Variables vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-INSTANCE-VARIABLES (instance method)

Format:

(SEND 'instance-name 'GET-INSTANCE-VARIABLES)

Description:

GET-INSTANCE-VARIABLES returns a list of symbol/value pairs for all instance variables associated with ***instance-name***.

GET-KIND-OF

Format:

(SEND 'classname '%GET-KIND-OF%)

Description:

GET-KIND-OF returns the contents of ***classname***'s Kind-Of vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-KINDS

Format:

(SEND 'classname '%GET-KINDS%)

Description:

GET-KINDS returns the contents of ***classname***'s Kinds vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-MEMBER-OF (class method)

Format:

(SEND 'classname '%GET-MEMBER-OF%)

Description:

GET-MEMBER-OF returns the contents of ***classname***'s Member-Of vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-MEMBER-OF (instance method)

Format:

(SEND 'instance-name 'GET-MEMBER-OF)

Description:

GET-MEMBER-OF returns the value of the Member-Of instance aggregation for ***instance-name***.

GET-MEMBERS (class method)

Format:

(SEND 'classname '%GET-MEMBERS%)

Description:

GET-MEMBERS returns the contents of ***classname***'s Members vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-MEMBERS (instance method)

Format:

(SEND 'instance-name 'GET-MEMBERS)

Description:

GET-MEMBERS returns the value of the Members instance aggregation for ***instance-name***.

GET-PART-OF (class method)

Format:

(SEND 'classname '%GET-PART-OF%)

Description:

GET-PART-OF returns the contents of ***classname***'s Part-Of vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-PART-OF (instance method)

Format:

```
(SEND 'instance-name 'GET-PART-OF)
```

Description:

GET-PART-OF returns the value of the Part-Of instance aggregation for ***instance-name***.

GET-PARTS (class method)

Format:

```
(SEND 'classname '%GET-PARTS%)
```

Description:

GET-PARTS returns the contents of ***classname***'s Parts vector position. For details on the specific contents of this class vector slot, the reader is referred to Appendix A, A Description of Galahad's Class and Instance Vectors.

GET-PARTS (instance method)

Format:

```
(SEND 'instance-name 'GET-PARTS)
```

Description:

GET-PARTS returns the value of the Parts instance aggregation for ***instance-name***.

MAKE-CLASS-ABSTRACT

Format:

```
(SEND 'METACLASS 'MAKE-CLASS-ABSTRACT
    'classname
    ['NOCOMPILE])
```

Description:

MAKE-CLASS-ABSTRACT is responsible for converting a previously created CONCRETE class to an ABSTRACT class. The resulting class can now no longer create instances or be added to an instance as an additional class. Subclasses of ***classname***, however, are not affected. CONCRETE subclasses still have the capability to generate instances.

Example:

```
(send 'metaclass 'create-class 'war-ship
      '(kinds submarine))

(send 'metaclass 'create-class 'submarine
      '(kind-of ((war-ship)))
      '(iv (maximum-depth feet (#!unassigned) (1 1))))

(send 'metaclass 'make-class-abstract 'war-ship)
```

In this particular example, we are converting the class WAR-SHIP from being a CONCRETE class to being an ABSTRACT class. After the above code is executed, no instances can be created from WAR-SHIP; however, SUBMARINE can still generate instances.

MAKE-CLASS-CONCRETE

Format:

```
(SEND 'METACLASS 'MAKE-CLASS-CONCRETE
      'classname
      ['NOCOMPILE])
```

Description:

MAKE-CLASS-CONCRETE is responsible for converting a previously created ABSTRACT class to a CONCRETE class. The resulting class can now have the capability of creating instances or being added as a new class to an instance. Subclasses of ***classname*** are not affected.

Example:

```
(send 'metaclass 'create-class 'war-ship
      '(kinds submarine)
      '(ABSTRACT))
```

```
(send 'metaclass 'create-class 'submarine
      '(kind-of ((war-ship)))
      '(iv (maximum-depth feet (#!unassigned) (1 1)))
      '(ABSTRACT))

(send 'metaclass 'make-class-concrete 'war-ship)
```

In this particular example, we are converting the ABSTRACT class WAR-SHIP to a CONCRETE class. After the above code is executed, WAR-SHIP can now create instances. SUBMARINE remains an ABSTRACT class.

RESET-GALAHAD

Format:

```
(RESET-GALAHAD)
```

Description:

This function returns the Galahad system to its initial starting configuration. All user-defined classes, simple classes, and instances are removed from the system.

Example:

```
(send 'metaclass 'create-class 'submarine
      (kind-of ((war-ship)))
      (iv (maximum-depth feet (#!unassigned) (1 1))))

(send ('submarine) '%create-instance%
      'submarine-1)

(reset-galahad)
```

In this example, SUBMARINE and SUBMARINE-1 are completely removed form the system.

SET-CLASS-VARIABLE-DEFAULT

Format:

```
(SEND 'METACLASS 'SET-CLASS-VARIABLE-DEFAULT
      'classname
      'classvar
      '(value . . . value)
      ['NOCOMPILE])
```

Description:

SET-CLASS-VARIABLE-DEFAULT is responsible for changing the default value of a class variable in a previously created Galahad class. Domain and cardinality constraints are enforced in the setting of this new default value.

Classname specifies the name of the class containing ***classvar***. ***(value . . . value)*** is the new default value(s) to be assigned to ***classvar***.

Example:

```
(send 'metaclass 'create-class 'submarine
    '(cv (number-in-fleet number (0) (1 1)
    '(iv (max-depth number (#!unassigned)(1 1)))))

(send 'metaclass 'set-class-variable-default
    'submarine
    'number-in-feet
    '(100))
```

In this example, we are changing the default value of Number-In-Fleet from 0 to 100. Galahad allows for this change in the default value because 100 passes the NUMBER domain constraint and the mandatory cardinality for this class variable.

SET-INSTANCE-VARIABLE-DEFAULT

Format:

```
(SEND 'METACLASS 'SET-INSTANCE-VARIABLE-DEFAULT
    'classname
    'instvar
    '(value . . . value)
    ['NOCOMPILE])
```

Description:

SET-INSTANCE-VARIABLE-DEFAULT is responsible for changing the default value of an instance variable in a previously created Galahad class. Domain and cardinality constraints are enforced in the setting of this new default value.

Classname specifies the name of the class containing ***instvar***. ***(value . . . value)*** is the new default value(s) to be assigned to ***instvar***.

Example:

```
(send 'metaclass 'create-class 'submarine
      '(cv (number-in-fleet number (0) (1 1)
      '(iv (max-depth number (#!unassigned)(1 1)))))

(send 'metaclass 'set-instance-variable-default
      'submarine
      'max-depth
      '(1000))
```

In this example, we are changing the default value of Max-Depth from #!UNASSIGNED to 1000. Galahad allows for this change in the default value because 1000 passes the NUMBER domain constraint and the mandatory cardinality for this instance variable.

START-GALAHAD

Format:

```
(START-GALAHAD)
```

Description:

The START-GALAHAD function begins the Galahad system and enables the user to communicate directly with the Galahad User Environment. Normally the only time this command is called is when the user accidentally falls into the Scheme inspector and needs to restart the system.

APPENDIX C

AN ANALYSIS OF TEXAS INSTRUMENT'S IMPLEMENTATION OF SCOOPS IN PC SCHEME

SCOOPS (Scheme Object-Oriented Programming System) is an object-oriented extension to TI Scheme, a specification developed by Texas Instruments for the Scheme programming language. One can describe SCOOPS as a language implemented in Scheme that provides the user with an object-oriented paradigm capable of modeling dynamic, lattice inheritance. The reader is referred to Chapter 5 of the TI Scheme Language Reference Manual and Chapter 3 of this paper for details about the salient characteristics of the SCOOPS language.

The purpose of this appendix is to provide the reader with an analysis of how SCOOPS was implemented in PC Scheme, a micro computer based implementation of the TI Scheme language specification. PC Scheme was developed by Texas Instruments for use in their Professional, Portable Professional, and BUSINESS-PRO line of micro computers. Also, the software engineers at Texas Instruments designed PC Scheme to operate on any 8088-based IBM or 100 percent compatible micro computer as well.

The information conveyed in this section is drawn from experimentation with several different SCOOPS applications and rather extensive use of PC Scheme's interactive debugger. The author also referred to the source code for an experimental, non production, version of the SCOOPS language. (Texas Instruments, 1986)

The designers of the SCOOPS language chose to implement SCOOPS classes as Scheme vectors and SCOOPS instances as environments. Consequently, the analysis presented below will center on these two data structures. Also, there will be a section at the end which describes each of the SCOOPS procedures and special forms and how they interact with both class vectors and instance environments. Unless otherwise noted, all references to a specific Scheme environment are to the lowest frame of that environment.

SCOOPS Class-Vector Description

SCOOPS represents object classes as Scheme vectors. When a class is created, SCOOPS builds a 15 slot vector to store all information about the class. As Figure C-1 illustrates, every class vector can be accessed from either the User Initial Environment or a SCOOPS class environment. In the User Initial Environment, the vector is bound to the user-supplied symbol in the DEFINE-CLASS statement. In a SCOOPS class environment, the vector is bound to the symbol %SC-CLASS. The details about each slot in a typical class vector are supplied below:

Vector Position 0

Label. Scoops Class Identifier

Format. "|#!CLASS|"

Description. This position is a label indicating the vector represents a SCOOPS class.

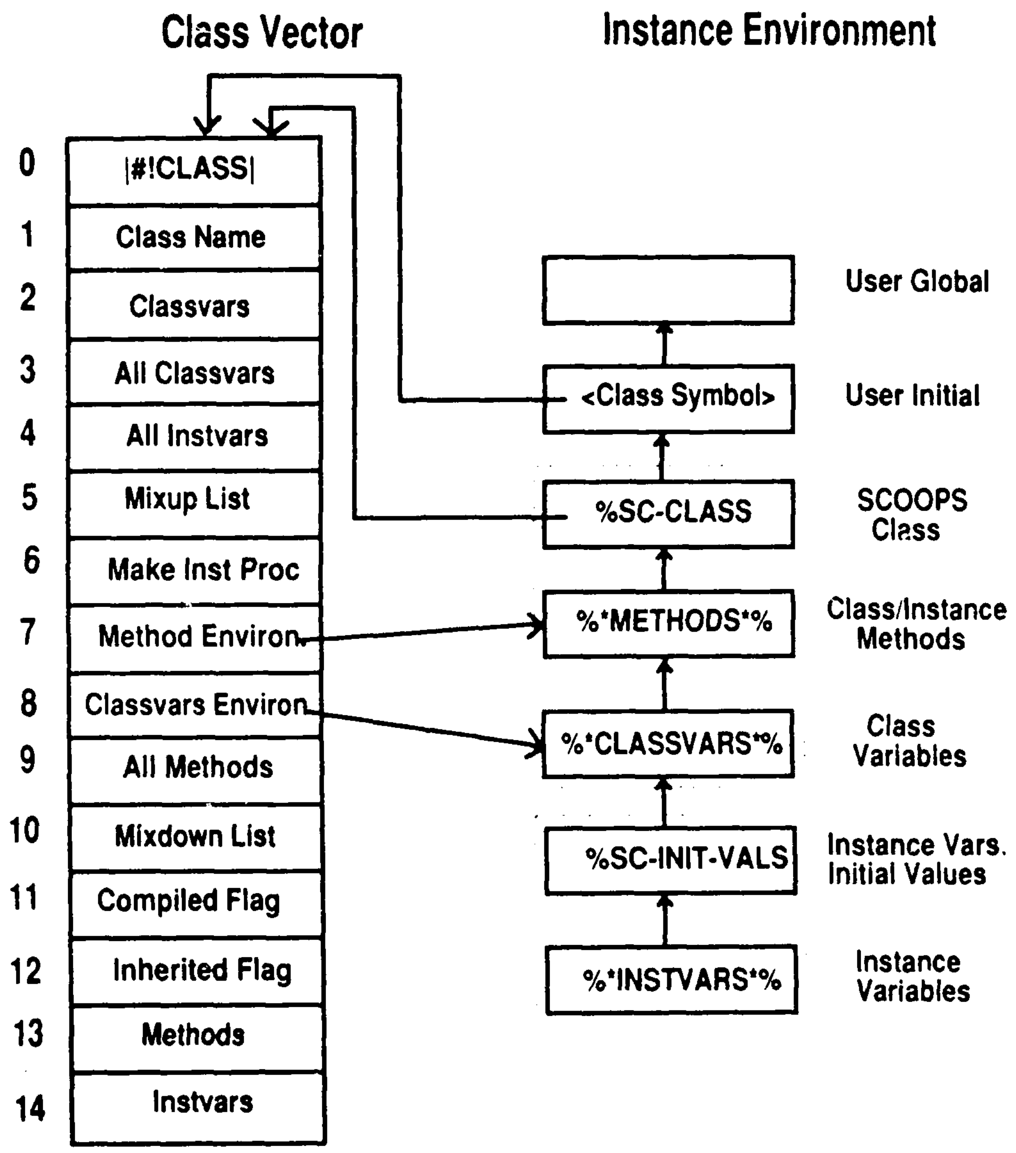


Figure C-1. SCOOPS Class Vector and Instance Environment Structure

Vector Position 1

Label. Class Name

Format. Classname

Description. The Class Name position identifies the name of the SCOOPS class represented by this particular vector. SCOOPS uses this position whenever it needs to retrieve the name of the class. The Class Name slot is assigned its value during the DEFINE-CLASS process.

Vector Position 2

Label. Class Variables

Format. ((Classvar Value)(Classvar Value)

. . .

(Classvar Value))

Description. This slot contains the symbol/value pairs of the locally defined class variables. SCOOPS uses the values in this position as a basis for beginning to build the All Class Variables slot (Vector Position 3). This element of the vector is assigned its value during the DEFINE-CLASS process.

Vector Position 3

Label. All Class Variables

Format. ((Classvar Value)(Classvar Value)

. . .

(Classvar Value))

Description. The All Class Variables slot contains the symbol/value pairs for both the locally defined and all inherited class variables. The COMPILE-CLASS function of SCOOPS uses this slot when building a class variables environment. COMPILE-CLASS assigns values to this position before it begins to build the class variables environment for the class.

Vector Position 4

Label. All Instance Variables

Format. ((Instvar (INITIALIZATION-PROCEDURE))
(Instvar (INITIALIZATION-PROCEDURE))
. . .
(instvar default))

Description. This slot contains the variable name/initialization procedure or variable name/default value for both the locally defined and all inherited instance variables. The initialization procedure is stored as source code and is present only for those variables declared to be INITTABLE. The procedure is replaced for non-INITTABLE variables with the variable's default value. A sample initialization procedure is as follows:

```
(APPLY-IF
    (MEMQ Instvar %SC-INIT-VALS)
    (LAMBDA (A) (CADR A))
    default value)
```

The initialization code in this slot is used by SCOOPS whenever there is a call to MAKE-INSTANCE. Initialization of instance variables takes place by evaluating code similar to that shown above in each newly created instance environment. The reader is referred to the next section, SCOOPS Instance Environments, for a description of SCOOPS instance environments and the role of the SCOOPS symbol %SC-INIT-VALS.

COMPILE-CLASS assigns the All Instance Variables position its value before the system determines the inheritance structure for the class.

Vector Position 5

Label. Mixup List

Format. (Classname . . . Classname)

Description. The Mixup List contains the names of the class's immediate parents in the inheritance hierarchy. SCOOPS reads this slot in left to right order to determine the class's inheritance structure and resolve conflicts for class variables, instance variables, and methods. This position is assigned its value based on the MIXIN clause of the DEFINE-CLASS statement.

Vector Position 6

Label. Make Instance Procedure

Format. Procedure Object

Description. The SCOOPS function MAKE-INSTANCE relies on the procedure contained in this position to return an instance environment, complete with instance variables and the variables' initial/default values. The Make Instance Procedure is also responsible for compiling the class if the class has not been previously compiled.

This slot is assigned its value during the DEFINE-CLASS process. It also changes to a different procedure object after the class has been compiled. The new procedure object no longer tests to see whether the class has been compiled. Instead, it returns an instance environment with all instance variables bound to their initial/default values.

Vector Position 7

Label. Method Environment

Format. An Environment Object

Description. This position points to the class's method environment. The pointer is used whenever SCOOPS needs to access this environment to add or delete methods. This slot is assigned its value during the COMPILE-CLASS process. The next section, SCOOPS Instance Environments, describes the method environment more fully.

Vector Position 8

Label. Class Variables Environment

Format. An Environment Object

Description. This position points to the class's class variable environment. The slot is used by SCOOPS to locate the environment whenever it needs to add, update, or delete class variables. The Class Variables Environment slot is assigned its value during the COMPILE-CLASS process. The SCOOPS Instance Environments section, describes the role of the class variables environment more fully.

Vector Position 9

Label. All Methods

Format.
```
(method-name
        (defining-class . inherited-class))
        (defining-class . inherited-class))
                        . . .
(method-name
        (defining-class . inherited-class))
```

Description. The All Methods position contains the names of both the locally defined and inherited methods of the class. COMPILE-CLASS is responsible for assigning this position its value and using this slot for building the methods environment. Also, DEFINE-METHOD and DELETE-METHOD update this position to reflect changes in the inheritance structure of user-defined SCOOPS methods.

The format of the All Methods position reveals how the system models dynamic lattice inheritance of SCOOPS methods. The key lies in the defining-class/inherited-class dotted pairs associated with each method. SCOOPS uses these dotted pairs to locate the actual procedure objects after inheritance conflicts have been resolved. The defining-class portion of the dotted pair is the name of the class containing the actual procedure object. The inherited-class portion tells SCOOPS from where the method was inherited in the inheritance network. To best understand the way this mechanism works, consider the inheritance network presented in figure C-2.

In this example, Class D contains the methods Method-1, and Method-2. By following SCOOPS' inheritance rules, the class inherits Method-1 from Class A and Method-2 from Class C. Method-1 is not inherited from Class B because of the inheritance heuristic of searching "left" superclasses in a depth-first search manner. The corresponding All Methods slot for Class D is as follows:

```
((method-1
      (class-a . class-a))
      (class-b . class-c))
            . . .
 (method-2
      (clas[illegible]ass-c))
```

Using [illegible]ll Methods slot above and the fact that methods are inherited first from Class A and then from Class C, SCOOPS can proceed to build pointers to the method procedure objects. All the system need do is access the CAR of the defining-class/inherited-class pair and copy the method pointer as it is stored in the defining-class vector (Vector position 13).

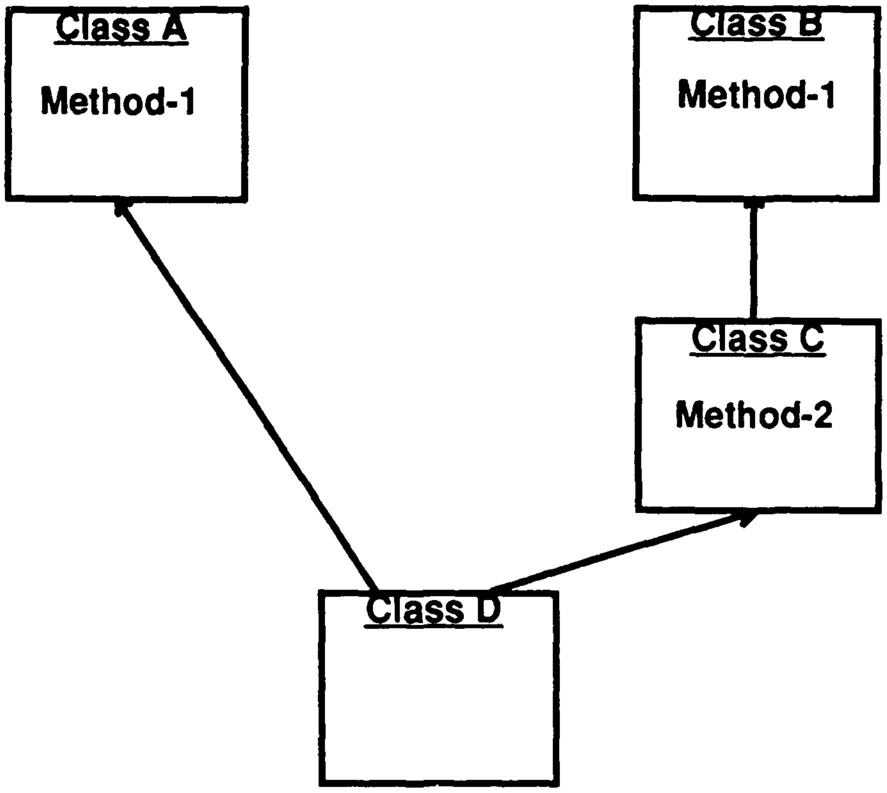


Figure C-2. SCOOPS and Lattice Inheritance for Methods

Notice the All Methods slot maintains information about methods that are not directly inherited by the class. For the example above, SCOOPS maintains information about Method-1 being stored with Class B. This is done because the inheritance of methods within the SCOOPS language is dynamic. A user could possibly delete Method-1 from Class A. In this case the system must know to retrieve the other Method-1 from Class B.

Vector Position 10

Label. Mixdown List

Format. (Classname . . . Classname)

Description. The Mixdown List contains the names of the class's immediate children in the inheritance hierarchy. SCOOPS uses this value to propagate newly created methods to a class's subclasses. This slot is assigned its value and updated during any COMPILE-CLASS of the class's immediate children.

Vector Position 11

Label. Compile Flag

Format. Boolean

Description. This is a flag indicating whether or not the class has been compiled. SCOOPS checks this flag to prevent itself from recompiling a class needlessly. It also refers to this flag in DEFINE-METHOD and DELETE-METHOD to determine if the method environment needs to be updated in addition to the class vector. The boolean value in this slot becomes TRUE only after the class has been compiled by either a direct call to the COMPILE-CLASS function or an automatic compile directed by MAKE-INSTANCE. The reader is referred to the <u>SCOOPS Primitives</u> section for details on MAKE-INSTANCE's automatic compilation of classes.

Vector Position 12

Label. Inherited Flag

Format. Boolean

Description. This is a flag indicating whether or not the class has been inherited by any of its subclasses. SCOOPS uses this flag to prevent itself from needlessly re-inheriting values from a class's parents. The boolean value in this slot becomes TRUE after either the class or one of its subclasses has been compiled.

Vector Position 13

Label. Methods

Format. ((method-name . procedure-object)

```
                    . . .
(method-name . procedure-object))
```

Description. This position contains only those methods locally defined for the class. When building a method environment, COMPILE-CLASS refers to this slot to obtain the actual procedure objects. DEFINE-CLASS assigns values to this position for class and instance variables that have been declared to be GETTABLE/SETTABLE. DEFINE-METHOD and DELETE-METHOD also update this slot to reflect any changes in the assignment of locally defined methods to the class.

Vector Position 14

Label. Instance Variables

Format. ((Instvar (INITIALIZATION-PROCEDURE))

```
(Instvar (INITIALIZATION-PROCEDURE))
                              . . .
(instvar default))
```

Description. This slot contains the variable name/initialization procedure or variable name/default value for instance variables that are defined locally within a class. The initialization procedure is stored in source code format and is present for only those variables declared to be INITTABLE. The procedure is replaced for non-INITTABLE variables with the variable's default value. A sample initialization procedure can be found with the description of Vector Position 4, All Instance Variables.

COMPILE-CLASS uses this slot as a basis for beginning to build the All Instance Variables position. This position is assigned its value during the DEFINE-CLASS process.

SCOOPS Instance Environment Frames

SCOOPS implements all instances as environments. An instance environment in SCOOPS consists of seven environment frames (see figure C-1). The top two frames, User Global Environment and User Initial Environment are standard among all PC Scheme applications. The next three frames, Scoops Class Environment, Methods Environment, and Class Variables Environment, are created when the instance's defining class is compiled. The bottom two frames, Scoops Initial Values Environment and the Instance Environment are build upon the creation of each individual instance. Figure C-3 illustrates the SCOOPS environment hierarchy when multiple classes with multiple instances are present.

SCOOPS evaluates all GET/SET and user-defined methods in the instance environment. By its evaluation in an instance environment, each method can directly access all symbols in the environment frame hierarchy. The description of a typical instance environment is presented below. We begin with the top frame, the User Global Environment.

User Global Environment

This environment contains Scheme primitives. Neither SCOOPS primitives nor SCOOPS application symbols reside in this environment.

User Initial Environment

In addition to all the SCOOPS primitives, this environment contains symbols bound to user-created classes and instances. The symbols are those supplied by the user during calls to DEFINE-CLASS and (DEFINE <symbol> MAKE-INSTANCE . . .) statements.

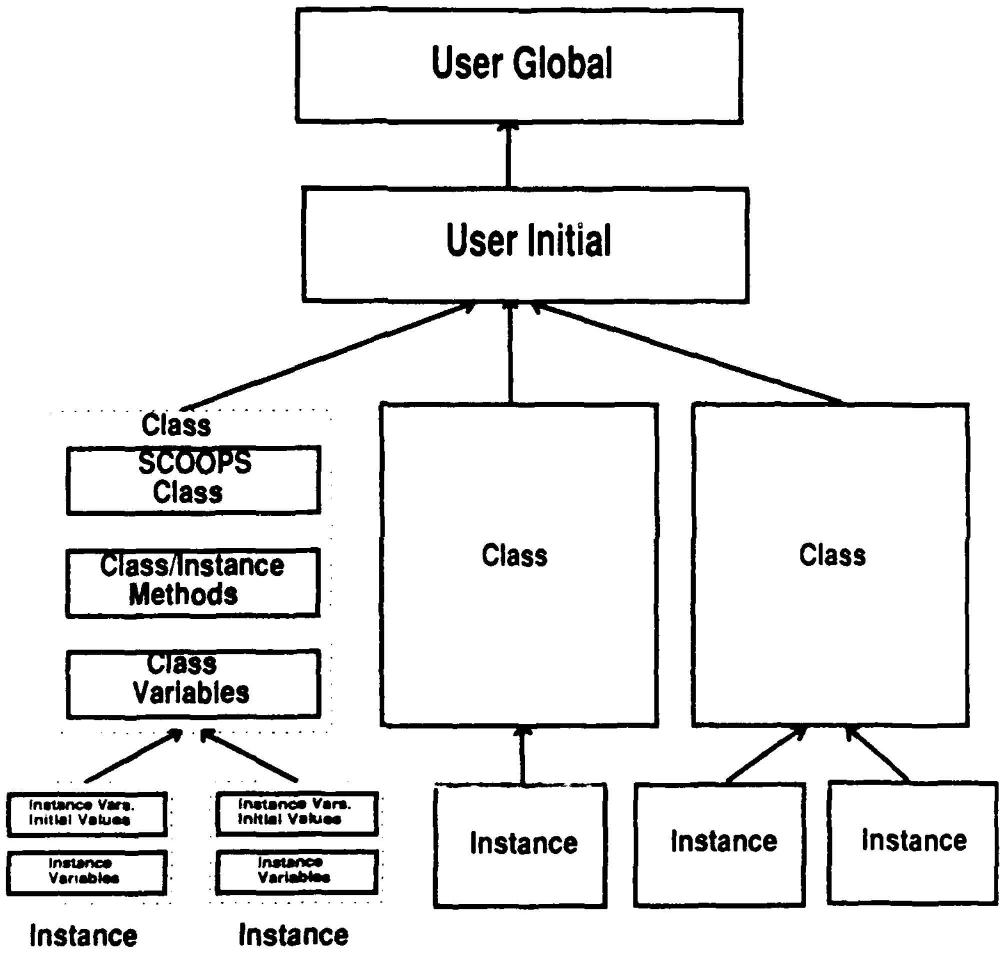


Figure C-3. SCOOPS Environment Hierarchy: Multiple Classes with Multiple Instances

Scoops Class Environment

This environment contains only the symbol %SC-CLASS. This symbol is bound to the vector representing the defining class of the instance. This vector is the exact same vector as referenced by the class's name in the User Initial Environment.

SCOOPS uses this environment and the standardized symbol %SC-CLASS for improving the run-time performance of the language. Any SCOOPS primitive evaluated in a lower environment can have immediate access to the class vector by referring to the symbol %SC-CLASS. This standardized form of access saves SCOOPS from having to reference a specific symbol when the system needs to refer only to an instance's defining class.

Methods Environment

This environment contains pointers to all locally defined and inherited methods in the SCOOPS class. Additionally, it contains the symbol %*METHODS*% which is always bound to the string: "-". The presence of %*METHODS*% is used as a documenting feature to label the environment as a SCOOPS methods environment.

The locally defined methods referenced in this environment are the exact same procedures as those identified in the class vector's 13th position. Additionally, all inherited methods are the same procedure objects as those contained in the super classes' 13th vector position as well.

This environment always reflects the current method structure of the SCOOPS class. The DEFINE-METHOD and DELETE-METHOD special forms update the Methods Environment to reflect any changes in the method inheritance structure.t

Class Variables Environment

This environment contains the bindings of all locally defined and inherited class variables of the SCOOPS class. Additionally, it contains the symbol %*CLASSVARS*% which is always bound to the string: "-". The presence of %*CLASSVARS*% is used as a documenting feature to label the environment as a SCOOPS class variables environment.

SCOOPS evaluates the SETCV and GETCV methods in this environment to access the current values of the class variables. Other methods can access these variables by their evaluation in the instance environment. The Class Variables environment always reflects the most current values of the class's class variables.

SCOOPS Initial Values Environment

This environment is built for each new instance created by the system. The environment contains only the symbol %SC-INIT-VALS. This symbol is bound to an ASSOC list containing instance variable/initial-value pairs. SCOOPS derives this list based on user input passed to the MAKE-INSTANCE special form. The instance variable initialization procedures, located in the class vector's fourth position, refer to the ASSOC list bound to %SC-INIT-VALS when assigning an initial value to an instance variable.

Instance Environment

This environment is the bottom level of the instance frame hierarchy. It contains both locally defined and inherited instance variables. Additionally, it contains the symbol %*INSTVARS*% which is always bound to the string: "-". The presence of %*INSTVARS*% is used as a documenting feature to label the environment as a SCOOPS instance environment. All user-defined methods and instance variable GET/SET methods are evaluated in this environment. By their evaluation in this environment, the methods can take advantage of Scheme's automatic traversal of the frame hierarchy to resolve externally referenced variables.

SCOOPS Primitives

After having examined the class vector and instance environment frame hierarchy, the reader can now more fully appreciate the internal workings of the SCOOPS procedures and special forms presented below. Each procedure/special form is described in terms of how it operates on the class vectors and instance environments.

ALL-CLASSVARS

This procedure examines the All Class Variables slot of the class vector (Vector Position 3), and returns the CAR of each variable/value pair.

ALL-INSTVARS

This procedure examines the All Instance Variables slot of the class vector (Vector Position 4), and returns the CAR of each variable/default-value pair.

ALL-METHODS

This procedure examines the All Methods slot of the class vector (Vector Position 9), and returns the CAR of each method/defining-class list.

CLASS-COMPILED?

This procedure examines the Compile Flag slot of the class vector (Vector Position 11) and returns its value.

CLASS-OF-OBJECT

This procedure is evaluated in an instance environment and returns the Class Name slot (Vector Position 1) of the vector bound to %SC-CLASS.

CLASSVARS

This procedure examines the Class Variables slot of the class vector (Vector Position 2), and returns the CAR of each variable/value list.

COMPILE-CLASS

This macro is responsible for completing the inheritance structure for each class and building the SCOOPS class, methods, and class variables environment frames. The specific tasks COMPILE-CLASS performs are as follows:

1. It establishes the compiling class's position in the inheritance hierarchy by adding the compiling class's name to the MIXDOWN slots (Vector Position 10) of the class's immediate parents. The compiling class's parents are derived from the class's MIXUP list (Vector Position 5).

2. COMPILE-CLASS updates those vector positions that require the traversal of the inheritance hierarchy for their completion. Specifically, these slots are: All Class Variables (Vector Position 3), All Instance Variables (Vector Position 4), and All Methods (Vector Position 9).

   In completing these vector positions, COMPILE-CLASS does not rely on the class's immediate parents having previously been compiled. In other words as COMPILE-CLASS searches for inheritable variables and methods, it does not reference the third, fourth, or ninth vector positions of any class. Instead, it examines only the vector positions of locally defined attributes. To ensure a complete inheritance structure, COMPILE-CLASS examines all super classes in the inheritance network.

3. It builds the necessary environments to complete the Method Environment slot (Vector Position 7) and the Class Variables Environment slot (Vector Position 8). To build the Method Environment, COMPILE-CLASS uses both the All Methods Slot (Vector Position 9) and the Methods slot (Vector Position 13). As previously mentioned, the All Methods position contains the method inheritance structure. The Methods slot contains the actual procedure objects.

4. COMPILE-CLASS modifies the Make Instance Procedure contained in Vector Position 6. The newly modified procedure, upon a call from MAKE-INSTANCE, returns a new instance environment complete with

instance variables bound to their initial/default values. The original procedure was responsible for compiling the class before returning an instance environment.

5. It sets both the Compile Flag (Vector Position 11) and the Inherited Flag (Vector Position 12) to TRUE.

Based on the comments above, the reader should note that COMPILE-CLASS does not automatically compile a class's subclasses. This feature is not necessary due to the way SCOOPS traverses the inheritance network.

DEFINE-CLASS

This macro is responsible for creating the class vector and assigning initial values to certain slots of the vector. A brief summary of a typical SCOOPS vector is presented below. The reader is referred to the SCOOPS Vector Description section of this appendix for details about each vector position.

Slot 0:	|#!CLASS|
Slot 1:	Class name
Slot 2:	Local Class Variables
Slot 3:	Local Class Variables (no inherited variables)
Slot 4:	Local Instance Variables (no inherited variables)
Slot 5:	Mixup List
Slot 6:	A "Uncompiled Make Instance" procedure
Slot 7:	Empty
Slot 8:	Empty
Slot 9:	Local Methods (no inherited methods)
Slot 10:	Empty
Slot 11:	Empty
Slot 12:	Empty
Slot 13:	Local Methods
Slot 14:	Local Instance Variables

DEFINE-METHOD

This macro is responsible for adding user-defined methods to SCOOPS classes. DEFINE-METHOD is unique in that it does not rely on a class's compilation to update the All Methods slot (Vector Position 9) of a class. Instead, it modifies the slot directly without going through the COMPILE-CLASS process. The specific functions DEFINE-METHOD performs are as follows:

1. It alters the source code of the method to allow the method to access externally referenced symbols in the current evaluation environment. Normally, externally referenced symbols are evaluated in the incrementally extended environment created when the procedure was defined (Eisenberg 129, 130). To best understand how the source code is modified, consider the following SCOOPS method: (Texas Instruments, 1987b, p. 7-153)

```
(define-method (employees earnings) ()
          (+ salary overtime))
```

   Salary and overtime are the externally referenced symbols in this method. DEFINE-METHOD takes the formals for this method (in this case the null list) and the body (+ salary overtime) and builds a LAMBDA special form. Additionally, to ensure SALARY and OVERTIME are evaluated in the current global environment, DEFINE-METHOD replaces references to these symbols with (EVAL <symbol>). For the method described above, the resulting special form is:

```
(lambda () (+ (eval salary) (eval overtime)))
```

2. DEFINE-METHOD evaluates the above special form and places the resulting procedure object in the defining class vector's Methods slot (Vector Position 13).

3. It updates the defining class vector's All Methods slot (Vector Position 9). For this particular class, both the defining class portion and the inherited class portion of the dotted pair are identical. This is due to the method being locally defined for the class.

4. DEFINE-METHOD then adds the newly created procedure object to the environment pointed to by the Method Environment slot (Vector Position 7) of the vector. The symbol bound to the procedure object is the name of the method as passed to the DEFINE-METHOD special form.

5. It repeats steps 3 and 4 for each class listed in the MIXDOWN list (Vector Position 10). For subclasses, the All Methods slot would reflect that the method was inherited and not locally defined in the class. Also, a subclass's Method Environment may not be updated due to SCOOPS's rules for the resolution of inheritance conflicts.

6. Step 5 is repeated recursively until the MIXDOWN lists of all subclasses have been exhausted.

DELETE-METHOD

This macro is responsible for removing user-defined methods from SCOOPS classes. DEFINE-METHOD erases user-defined methods by replacing the method's original procedure object with a SCOOPS primitive that identifies the method as having been deleted. This way, DELETE-METHOD does not have to restructure the method inheritance hierarchy. Instead, upon being called by the user application, the SCOOPS primitive merely reports the method as having been deleted. DELETE-METHOD modifies the Methods slot (Vector Position 13) of the method's defining class. The special form also updates the procedure object pointed to by the method name in the Methods environment.

DESCRIBE

This procedure prints selected information about a SCOOPS class or instance. This primitive first tests to see whether the object to be described is either a class (a vector) or an instance (an environment). If the object is a class, DESCRIBE formats and prints the contents of the following vector slots:

Slot 1:	Class Name
Slot 3:	All Class Variables
Slot 4:	All Instance Variables
Slot 5:	Mixup List
Slot 9:	All Methods
Slot 11:	Compiled Flag
Slot 12:	Inherited Flag

If the object is an instance, DESCRIBE then formats and prints the symbol/value pairs of the class variables in the Class Variables environment and the Instance environment.

GETCV

This macro retrieves the value of a SCOOPS class variable. GETCV first checks to see if the class has been compiled. It does this by examining the Compiled Flag slot of the class vector (Vector Position 11). If the class has already been compiled, GETCV then evaluates the GET method associated with the class variable. Evaluation occurs in the Class Variables environment. If the class has not been compiled, GETCV issues an error message to the user.

INSTVARS

This procedure examines the Instance Variables slot of the class vector (Vector Position 14), and returns the CAR of each symbol/value pair.

MAKE-INSTANCE

This macro is responsible for building both the SCOOPS Initial Values and Instance environments. Within the Initial Values environment, MAKE-INSTANCE binds %SC-INIT-VALS to the ASSOC list containing the initial values of the instance variables. After binding %SC-INIT-VALS, MAKE-INSTANCE then evaluates the procedure located in the Make Instance slot (Vector Position 6). This procedure is responsible for returning a new instance environment with the instance variables bound to their appropriate values.

METHODS

This procedure examines the Methods slot of the class vector (Vector Position 13), and returns the CAR of each symbol/procedure-object pair.

MIXINS

This procedure examines the MIXUP List slot of the class vector (Vector Position 5), and returns the symbols found in the slot.

NAME->CLASS

This macro searches for a match between the symbol passed to NAME->CLASS and the Class Name slot (Vector Position 1) of all SCOOPS classes defined in the system. If a match occurs, this special form returns the class vector of the corresponding class.

RENAME-CLASS

This macro modifies the Class Name slot (Vector Position 1) of the class vector. It also adds the class's new name as another symbol in the User Initial Environment. The symbol corresponding to the old name of the class remains and continues to point to the newly updated vector.

SEND

This macro builds a procedure call with the selector name and parameters passed in the SEND statement. It then evaluates the resulting procedure call in the Instance Environment corresponding to the SEND's destination. The result of the function call is returned to the user.

SEND-IF-HANDLES

This macro first examines the Methods environment of the destination instance's defining class. If the selector passed to the SEND exists as a method in the Methods environment, SEND-IF-HANDLES builds the function call list and evaluates the method. Otherwise, the macro returns nil.

SETCV

This macro assigns a value to a SCOOPS class variable. SETCV first checks to see if the class has been compiled. It does this by examining the Compiled Flag slot of the class vector (Vector Position 11). If the class has already been compiled, SETCV then evaluates the SET method associated with the class variable. Evaluation occurs in the Class Variables environment. If the class has not been compiled, SETCV issues an error message to the user.