

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

FORMAL METHODS FOR REUSE OF DESIGN
PATTERNS AND MICRO-ARCHITECTURES

SRIDHAR NARAYANAN

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

OCTOBER 1996

© SRIDHAR NARAYANAN, 1996



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-26018-6

Canada

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Sridhar Narayanan**

Entitled: **Formal Methods For Reuse Of Design Patterns And
Micro-Architectures**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards
with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
_____ Examiner
_____ Examiner
_____ Examiner
_____ Supervisor
_____ Co-supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 19 _____
Dean of Faculty

Abstract

Formal Methods For Reuse Of Design Patterns And Micro-Architectures

Sridhar Narayanan

Software reuse is recognized to have the potential for improved productivity of quality software. Class reuse, micro-architecture reuse, and reuse of application frameworks are the three distinct levels of software reuse. This thesis examines the critical issues in providing support for different levels of reuse through formal specifications of reusable components.

The formal specification language Larch/C++ has been used in reuse research projects and has been found to be adequate for specifying the behavior of class interfaces. We strengthen this claim by applying this methodology to several classes chosen from Rogue Wave library. We then provide extensions to the language for specifying object collaborations in a micro-architecture. We then illustrate this specification methodology for a micro-architecture chosen from a design pattern repository. In this context, we have also discussed the need to formally document design patterns and provided a formal framework within which design patterns can be formalized. Finally, we present the conceptual schema for a design pattern repository and discuss the query language features for storage and retrieval of design patterns.

Acknowledgments

I wish to express my deepest gratitude to my supervisors Professor V.S. Alagar and Professor R. Missaoui. Their sustaining guidance and encouragement made my thesis work a very pleasant and educational experience. I am also thankful to them for the financial support.

I also thank my parents and my uncle and aunt in Canada for their valuable support and encouragement.

Contents

List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Benefits of Reuse	1
1.2 Object-Oriented Systems and Reuse	2
1.3 Scope of the Thesis	6
2 Basic Concepts - Design Patterns, Micro-architectures and Levels of Reuse	8
2.1 Design Patterns	8
2.2 Frameworks	12
2.3 Levels of Reuse	13
2.3.1 Reuse-In-The-Small	13
2.3.2 Reuse-In-The-Medium	15
2.3.3 Design Pattern Reuse	17
2.3.4 Pattern Repositories	20
2.3.5 Other Levels of Reuse	21
3 Specifying C++ classes in Larch/C++	22
3.1 Larch/C++ - A Quick Overview	25
3.1.1 Larch Shared Language - LSL	25
3.1.2 Larch/C++	26
3.2 Improvements on Larch/C++	30
3.2.1 Composite Sorts	30
3.2.2 The Formal Model of Objects, Values, and States	31

3.2.3	Declarations and Declarators	32
3.2.4	State Functions	33
3.2.5	New features in Function Specifications	34
3.2.6	New Features in Class Specifications	38
3.3	Motivation for using Rogue Wave Software	40
3.4	Interface Specifications of Rogue Wave Classes	41
3.4.1	Interface Specification of RWOrdered	42
3.5	Experience in formalizing Rogue Wave library classes	48
3.5.1	Ambiguities and Inconsistencies	48
3.5.2	Reuse of LSL Traits	50
4	Specifying Object Collaborations - A Formal Approach	53
4.1	Layer 1 – LSL Traits	54
4.2	Layer 2 – Role Specification	54
4.2.1	V-action, S-action, O-action.	55
4.3	Layer 3 – Collaboration Specification	56
4.4	A Clock Example	59
4.4.1	Problem Description	59
4.4.2	Documenting a Solution	61
4.5	Specification of the Case Study	62
4.5.1	Layer 1 Specification	62
4.5.2	Layer 2 Specification	67
4.5.3	Layer 3 Specification	69
5	Formalizing Design Pattern Documentations	70
5.1	Describing Design Patterns - An overview	70
5.1.1	Describing Patterns - GOF Style	71
5.1.2	Describing Patterns - Doug Schmidt's Style	79
5.1.3	Comparison of the Two Styles	80
5.2	Template - A New Documentation Style	83
5.2.1	Abstract Factory	85
5.2.2	Composite	86
5.2.3	Observer	88
5.3	Towards A Formal Approach	91

5.3.1	Problem Section	91
5.3.2	Constraints Section	93
5.3.3	Reuse Requirement Section	94
5.3.4	Solution Section	95
5.3.5	Reuse Requirements Resolution Section	96
5.3.6	Drawbacks Section	97
5.3.7	Summary	97
5.4	A Formal Approach	98
5.4.1	Sorts	98
5.4.2	Class Level Predicates	100
5.4.3	Object Level Predicates	102
5.4.4	Type Predicates	104
5.4.5	Other Predicates	104
5.4.6	Interface and Interaction Specifications	105
5.5	Applying the Formalism	107
5.5.1	Formal Template	107
5.6	Abstract Factory	109
5.6.1	Problem	109
5.6.2	Constraints	110
5.6.3	Solution	110
5.6.4	Reuse Requirements	112
5.7	Composite	112
5.7.1	Problem	112
5.7.2	Constraints	113
5.7.3	Solution	113
5.7.4	Reuse Requirements	114
5.8	Observer	114
5.8.1	Problem	114
5.8.2	Constraint Formalism	115
5.8.3	Solution	115
5.8.4	Reuse Requirements	117
6	Design of a Pattern Repository	118
6.1	Motivation	118

6.2	Problem	120
6.2.1	Inter Design Pattern Queries	121
6.2.2	Intra Design Pattern Queries	123
6.3	Choosing a Database Management System	124
6.4	Towards the Design	127
6.5	The Conceptual Schema	128
6.6	Schema Definitions	134
6.6.1	Integrity Constraints	137
6.6.2	An Example	138
7	Conclusions	140
	Bibliography	142

List of Figures

1	Larch/C++ Model of a List Object	31
2	Import List Specification of a few classes	42
3	LSL Traits in the Specification of Ordered Vector	43
4	Reuse of LSL traits	50
5	MasterClock – ZonalClock Micro-Architecture	59
6	Design of the Micro-architecture	60
7	Object Collaboration Graph for the Clock Example.	60
8	AbstractFactory - Contextual Example	73
9	OMT Diagram For Abstract Factory Structure	75
10	OMT Diagram For Composite Structure	87
11	Composite - Contextual Example	88
12	OMT Diagram For Observer Structure	90
13	Observer - Contextual Example	90
14	High Level Object Model For Database Schema	130
15	Object Schema for the Design Pattern Solution	130

List of Tables

1	Sorts of Global Variables	52
2	Sorts of Formal Parameters	52

.

Chapter 1

Introduction

The concept of software reuse was introduced in a seminal paper by McIlroy [47] at the 1968 Nato Software Conference – the founding date of Software Engineering discipline. Recognizing the most obvious benefits of software reuse, such as productivity gain and improved software quality, many commercial, government and industrial organizations in the USA, Japan, and Europe have recently instituted systematic software reuse programs [42]. There are numerous diverse technical and non-technical issues challenging the wider practice of software reuse. Among them, the most important technical issue is the use of specification languages for creating, labelling and retrieving software components for error-free reuse.

1.1 Benefits of Reuse

Software Development is becoming increasingly capital-intensive, tool dependent, cooperative, and requires greater early investment of capital in return for reduced cost at later stages. Thus reusable software is to be viewed as a capital good whose investment cost is recoverable amortized over a number of years from its large pool of users. The additional expected benefits of reuse are:

- It can lead to improved reliability and performance.
- Reuse promotes interoperability between systems.
- It supports rapid prototyping.

- It provides a transparent uniformity among systems constructed with the same components.

Recent reports from industries such as Hewlett-Packard[42], NEC and Fujitsu [8, 23], Digital[19] and IBM[24] also suggest that there is a reduction in defective software, increase in return on investment, improved efficiency in product delivery and in general, a corporate-wide acceptance of reuse. Hence it can be argued that systematic application of reuse to prototyping, development, and maintenance during software development process is an effective way to reduce cost and improve software reliability and productivity.

1.2 Object-Oriented Systems and Reuse

Object-oriented (OO) approaches to design of software systems have proven to significantly improve software quality and the underlying development effort.

Object-Oriented programs are made up of objects. An object packages both data and procedures that operate on the data. The procedures are typically called methods or operations. An object performs an operation when it receives a request (or message) from a client. Requests are the *only* way to get an object to execute an operation. Operations are the *only* way to change an object's internal data. Because of these restrictions the object's internal state is said to be *encapsulated*; it cannot be accessed directly, and its representation is invisible from outside the object. Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as operation's *signature*. The set of all signatures defined by an object is called *interface* of the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object. The sender of a message does not need to know how the message is processed internally by the object, only that it responds to particular message in a well-defined way. Thus, from the point of view of an object's clients only the object's interface behavior is important.

An object's implementation is defined by its class. Classes are the basic software modules in an object-oriented design. Objects are created by *instantiating* a class. The object is said to be an *instance* of a class. The process of instantiating a class allocates

storage for the object's internal data (made up of instance variables) and associates operations with these data. One of the cornerstones of object-oriented programming is *data abstraction* and refers to the fact that a class's interface is completely independent from its implementation. Classes implemented by one programmer (the *producer* or *implementor*) can be used by other programmers (the *clients*) without these clients having to concern themselves with understanding a class's implementation details. The client of a class needs only to understand the behavior of the class as specified by the method interfaces and may view the method implementations as being contained in a *black-box* hidden from view. This approach is conducive to reuse, because a module can be used by a client without concern for the implementation details of this module. There is a considerable amount of saving of time and cost – much less effort is required than to write a module from scratch or to reuse an existing module after understanding its implementation details.

New classes can be defined in terms of existing ones using class inheritance. When a subclass inherits from a parent class it includes the definition of all data and operations that the parent class defines. Subclasses can refine and redefine behaviors of their parent classes. More specifically a class may override an operation defined by its parent class. Overriding gives subclasses a chance to handle requests instead of their parent classes (for e.g., virtual functions in C++). Class inheritance lets the designer define classes simply by extending other classes, making it easy to define families of objects having related functionality. Yet, there is no agreement among the object-oriented community on the notion of inheritance, it seems to encompass the following distinct concepts [25, 61, 44].

- **Implementation Inheritance:** This is a mechanism for the incremental definition of new classes based on existing ones. It can be defined as the carrying of features (methods and instance variables) from a parent class definition to its child class and the possible overriding of methods in the child class. By sharing implementation code which describes the internal representation of classes, the total amount of code in a system can be reduced drastically. In general, this form of inheritance does not provide any guarantees that the newly derived class will be a specialization of its parent class. This is because a method may be overridden in the derived class in a way that is not consistent with the parent class, or a subclass may even hide certain signatures provided in the superclass.

- **Subtype Inheritance:** This form of inheritance is of two types:
 - **Interface Inheritance:** In this form of inheritance the subclass inherits the interface of the parent class. An object's type refers here to its interface— the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type. There is a close relationship between the class and type. Because a class defines the operations an object can perform, it also defines the object's type. When we say an object is an instance of a class, we imply that the object supports the interface defined by a class. In this form of inheritance, an object of the subclass provides the interface of the parent class and can therefore be substituted for a parent class object in a program. However reliability of such a substitution cannot be assured because a method may be overridden in a derived class in a way which is not consistent with the parent class. The subclass is referred to as a weak subtype of the parent class supertype.
 - **Behavior Inheritance:** This is a relationship between the specification of two classes. This form of inheritance includes interface inheritance but requires that the behavior of the inherited signatures in a subclass stay consistent with the behavior in the superclass. This subtyping property captures the notion of behavior compatibility between two classes by requiring that members of a subtype are also members of supertype. The subtyping property ensures that a subtype can be reliably substituted for a supertype in a specification or a program.

One of the key features of object-oriented programming (OOP) which facilitates software reuse is the support for polymorphism, as provided by its message passing (or late binding) paradigm. The type of polymorphism provided in OOP has been extensively studied and characterized as *inclusion polymorphism* [9] or subtype polymorphism [40, 41]. It has been shown that this type of polymorphism, popularized by its ability to support code reuse in Smalltalk-80 [28], is fundamentally different from the *universal parametric* polymorphism found in functional languages such as ML [9, 39, 20].

Subtype polymorphism is distinguished from other kinds of polymorphism by two

features: (i) the dynamic binding of operation names to operations based on the run-time types of their arguments, and (ii) the possibility that a given expression may denote objects with different types (ie. different subtypes) at run-time [41]. These properties allow programs which make use of subtype polymorphism to abstract over a set of heterogeneous objects that have similar behavior. This makes it possible to extend, or customize, an existing program by introducing new types of objects on which the program can operate. In turn, this ability to easily extend a program is very conducive to software reuse. When such an extension is made the existing program is entirely reused, without modification, and only implementations of new types need to be added.

Polymorphic code in OOP can manipulate objects of several different types, provided the actual (dynamic) type of an object conforms to the the static (or nominal) type of the expression that denotes it. This capability has important consequences on the ability to support reuse. It means that new subtypes can be easily be added to a system without modifying existing generic (polymorphic) modules. This is to be contrasted with non-polymorphic typed languages (eg. C, Pascal, Modula) where the addition of new types typically involves the addition of a “type tag” on the new data type, and the verification of this type tag in a “case” statement. This latter approach is very dangerous and error-prone, as it is easy to forget updating one of the potentially many case statements. This leads to software that is difficult to reuse and maintain [17]

In fact, it has been argued very convincingly that subtype polymorphism is essential for the development of extensible, reusable software components [17, 18, 48]. The non-polymorphic approach to code reuse make black-box reuse of code impossible, since to make an extension to a system (ie. to add a new type) it is necessary to modify existing code as well as add new code [17]. In the subtype polymorphic OO approach, existing polymorphic code can be left intact since it is sufficiently generic to accommodate extensions.

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. Class inheritance allows one to define the implementation of a class by reusing the implementation and functionality of existing classes. Thus, the time and effort needed to define new kinds of objects are drastically reduced. However implementation reuse is only half the story. Inheritance’s ability to define families of objects with identical interfaces is also important,

since polymorphism depends on it. In subtype inheritance all classes derived from an abstract class will share its interface. All subclasses can then respond to requests in the interface of an abstract class, making them all subtypes of abstract class. Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. Because, objects are accessed solely through their interfaces, we do not violate encapsulation. Further, due to polymorphism an object can be replaced at run-time by another as long as it has the same type. Since, an object's implementation is written in terms of object interfaces, there are substantially fewer implementation dependencies.

It is thus clear that OO design concepts and OO programming languages offer a high potential for reuse.

1.3 Scope of the Thesis

The thesis makes four distinct contributions to reusing software designed by OO principles:

- Building on the seminal work of Colagrosso [16], Larch/C++ specifications for some more Rogue Wave library *tools.h++* classes are given and our experiences are summarized. This work is presented in Chapter 3.
- Larch/C++ language is extended with constructs for expressing object collaborations. This extended three-tiered language is used to specify micro-architectures. This work, the first of its kind, appears in Chapter 4.
- A critique on existing design pattern documentations, a rationale for a new documentation, and a formalism for essential features of design patterns are given in Chapter 5. Once again, this is the first time any research is directed at formalizing design pattern documentations.
- Software reuse in large scale is feasible when software artifacts are stored and retrieved from a database. The thesis presents an object-oriented database schema, and possible queries that can be posed on the database of design patterns in Chapter 6. There is no published work addressing this issue.

A brief summary of basic concepts on design patterns, micro-architectures, levels of reuse are discussed in Chapter 2. The thesis concludes with a summary and directions for future research in Chapter 7.

Chapter 2

Basic Concepts - Design Patterns, Micro-architectures and Levels of Reuse

In this chapter we provide a brief summary of basic concepts on design patterns, micro-architectures, levels of reuse that can be practised in developing OO software systems.

2.1 Design Patterns

In spite of the high potential of reuse offered by object-oriented design concepts, designing reusable object-oriented software is an extremely challenging task. The designers must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationship among them. The design should be specific to the problem at hand, but also general enough to address future problems and requirements [33, 25]. The designers want to avoid redesign or at least minimize it. A reusable and flexible design is difficult if not impossible to get right the first time. So, expert designers often reuse solutions to design problems that they have discovered in the past instead of solving the problem from scratch. In fact, many object-oriented software systems contain such recurring patterns of classes and communicating objects that solve specific design problems and

make such designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them. Such patterns are called **design patterns**.

Object-oriented software community is still divided on offering a precise definition of what constitutes a design pattern. Several definitions have been proposed. Gamma et al. [25] define them as descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. It is clear that design patterns are abstract representation of solutions applied repeatedly to problems that arise when developing object-oriented software within a particular context. Besides, capturing the static and dynamic structure and collaborations among the key participants in software designs, design patterns also document the intent, rationale, and consequences of applying the abstract design in various contexts. Each design pattern focuses on a particular object-oriented design problem or issue. It describes where it can be applied, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use [25].

According to Gamma et al. [25] a design pattern has four essential elements:

- The **pattern name** is used as a handle to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary and makes it easier to think about designs and to communicate them and their trade-offs to others.
- The **problem** describes when to apply the pattern. It explains the problem and its **context**. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of the design solution and how a general arrangement of classes and objects solve.

- The **consequences** are the results and trade-offs of applying the pattern. Consequences are often unvoiced when we describe design decisions. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps one understand and evaluate them.

The documentation style used in [25] uses textual description and graphical notations for describing patterns. We discuss these in detail in Chapter 5.

As an example, let us consider the Model/View/Controller (MVC) triad of classes [36], which is used to build user interfaces in Smalltalk-80. MVC contains three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to its user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse. The MVC contains design pattern Observer that decouples model and views in such a way that multiple views can be attached to a model to provide different interpretations. One can also create new views for a model without rewriting it. MVC contains Composite pattern that represents nested views structures in such a way that Composite view objects act just like view objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views. MVC also contains Strategy pattern that represents the View-Controller relationship. MVC encapsulates the response mechanism (to user input in the view) in the Controller object. A view uses an instance of the Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller.

Design patterns thus offer a very high potential of reuse and have the following advantages over the reuse of stand alone classes [54]:

- Design patterns enable large-scale reuse of software architecture. Instead of building an object model from class templates, the designer is able to use pre-defined groupings (patterns) of class templates.
- Design patterns explicitly capture the expert knowledge and design trade-offs. They help the designer choose design alternatives that make a system more reusable and avoid alternatives that compromise reusability.

- They help improve developer communication because they provide developers with a shared vocabulary and concepts.
- Expressing proven techniques as design patterns makes them more accessible to developers of new systems.
- Design patterns can even improve documentation and maintenance of existing systems by providing explicit specification of class and object interactions and their underlying intent.
- Design patterns help solve many of the common problems object oriented designer face by helping [25]:
 - identify less obvious abstractions and objects that can capture them
 - determine object granularity
 - define an interface by identifying what must and must not be part of interface, the kinds of data that gets sent across the interface
 - determine the type of inheritance required in a problem
- Design patterns emphasize the use of sound object-oriented concepts. All the patterns in Gamma et al [25] have been designed using the principles:
 - *program to an interface, not to an implementation* and
 - *favoring object composition over inheritance*

The reasons for such design principles are:

- Manipulating the objects solely in terms of the interfaces of abstract classes results in clients remaining unaware of the specific types of objects they use, as long as the objects adhere to interfaces the clients expect and clients also remain unaware of the classes that implement these objects. Clients only know about the classes defining the interface.
- Unlike object composition, implementations inherited from parent classes cannot be changed at run-time, because inheritance is defined at compile time. Secondly, parent classes would define at least part of their subclass' physical representation. Because inheritance exposes a subclass to details

of its parent's implementation, it is often said that "inheritance breaks encapsulation" [58].

2.2 Frameworks

Frameworks represent generative architectures designed for reuse. They exist in a certain application domain, and provide implementation of an architecture from which a family of applications in that domain can be derived. The main architectural components of a framework are abstract classes, which define the role and protocol of the components, and their behavior as defined by their collaborations. Frameworks not only provide all the basic functionality required for a given application domain, but also the flexibility of being able to customize and refine most of this functionality to suit the needs of a particular application. Examples of popular frameworks are the ET++ – a graphical user interface application framework [60] and HotDraw [34].

The potential for reuse of frameworks greatly exceeds that of class libraries. Whereas class library components are used individually, frameworks allow reuse of the abstract design of an entire application.

Frameworks and design patterns are both meant to be reused. However, in spite of their similarities they differ in the following ways [25]:

- Since, frameworks are embodied in code, they can be reused as is, but design patterns represent abstract designs and must be implemented each time they are used.
- Unlike frameworks, design patterns explain the intent, trade-offs and consequences of design.
- Design patterns are smaller architectural elements than frameworks. A typical framework consists of several design patterns but the reverse is not true. Instantiations of patterns within frameworks are referred to as micro-architectures.
- Frameworks always have a particular application domain. A graphical editor framework might be used in a factory simulation, but may not be mistaken for simulation framework. However design patterns can be reused in several applications.

- Unlike design patterns, a framework dictates an application's architecture.

Frameworks and design patterns are exciting developments in the object-oriented technology and hold the potential for making large scale software reuse a reality.

2.3 Levels of Reuse

Object-oriented software components can be reused at three different levels of abstractions [37].

2.3.1 Reuse-In-The-Small

The first level is *reuse-in-the-small*, which involves the reuse of a *class* or a *method*, and/or a *code* fragment. In this thesis we are interested in the black-box reuse of classes. A class serves as a reusable component in an object-oriented system that can be reused either through inheritance or through object composition. In either case, black-box reuse is the preferred way of reusing a class since in this form of reuse clients manipulate the objects in the class only through its interface and are not dependent on the implementation provided by the class. The black-box approach to reuse assumes that a class's behavior can be succinctly described without having to refer to an implementation. OO programming languages offer no support for this ability. Instead, programmers must rely on the informal comments in class interface files to understand the semantics of a class. However, their imprecise, verbose, and potentially ambiguous and incomplete nature often prevents them from being of much help [48, 63, 31]. A natural language description is not sufficiently precise to describe the exact functioning of a software module as a black-box, and leads to ambiguities and differences of interpretation from one programmer to the other. What appears obvious to the implementor of a class may not be so obvious to a client that is supposed to rely on nothing more than the class's method signatures and informal comments on how to use it.

The result of this inability to adequately specify the behavior of reusable components is that programmers are forced to turn to the implementation to fully understand its behavior. However, the sheer volume of existing classes, their complex interactions, and their implementation details frustrate programmers who by inspecting the code, must try to understand the behavior of potentially useful classes.

The net effect of all this is that the client may ultimately make improper use of a class, resulting in a defective program. The client's only avenue to solve the program defect is then often to "debug" the code by tracing its execution to try to determine the cause of the interfacing (ie. usage) problem. As a result, a substantial amount of time is spent trying to learn how to use a class correctly. All this wasted effort offsets part or all of the productivity increases which should have resulted by reusing the module rather than rewriting it.

The above observations indicate the need for a formal semantic definition of reusable class interfaces. Because of its precision and correctness, a formal specification language has the potential of overcoming the problems associated with the use of natural languages for specifying the semantics of reusable OO components.

Formal specifications also have another advantage over natural language in that they can be automatically processed. This opens the door for such things as automatic searching in component libraries, specification syntax checkers, and partially automated semantic analysis of specifications [57, 63, 21]. Because formal specifications are mathematical objects, it is also possible to develop formal criteria and algorithms to evaluate properties of these specifications such as completeness and consistency.

The formal specification can be used as a basis for organizing the classes according to a specification hierarchy. The specification language itself can incorporate the concept of hierarchy in such a way that specifications of the supertype are inherited by a subtype and need not be repeated in the subtype. This allows reuse at the level of specifications, in addition to reuse at the level of code, so that new specifications can be developed incrementally based on existing specifications.

Polymorphic programs that use message passing can be difficult to reason about, because the effect of a *message send* depends upon the type of the *receiving* object. There may be many different operations that could be executed by a *message send* and the same piece of code may result in the execution of different method implementations during different executions. One approach to reasoning about polymorphic programs would be to perform an exhaustive case analysis by considering all possible object types that a message selector can involve. However, this approach is impractical in large systems. This approach also has a severe disadvantage that adding a new type of object to a system can require additional case analysis of message invocations in existing polymorphic code. To obtain the advantage of extensibility promised by

object-oriented methods, unchanged program modules should not have to be respecified or reverified when new types of objects are added to a program. Since one does not have to update the code (because of late binding), it would be tiresome if one had to reverify the implementation of existing polymorphic code. Leavens [39, 40, 41] has done work in this direction and has arrived at a modular specification and verification technique for OO programs which makes it unnecessary to respecify and reverify unchanged polymorphic code when new types are added. His verification technique is sound and complete and can be used to formally verify the correctness of implementation involving polymorphic code with respect to its specification.

In this thesis we build on the work of Collogrosso [16] and illustrate the use of formal specification language Larch/C++ to specify C++ classes from the Rogue Wave Library tools.h++ [51].

2.3.2 Reuse-In-The-Medium

The next level of reuse is *reuse-in-the-medium*, the reuse of classes and their interactions. We shall refer to this type of reuse as the *micro-architecture*¹ reuse. This level of reuse itself can occur at two levels of abstraction viz. the reuse of design patterns in the design of frameworks and applications and reuse of frameworks and applications themselves. We shall refer to the former as design pattern reuse and the latter as micro-architecture reuse (even though it is not necessary that every grouping of class and their interactions represents an instantiation of a design pattern).

An application or framework is composed of more than just classes. A complex system requires many levels of abstraction one nested within another. Classes are a way of partitioning and structuring an application for reuse. But designs often have groups of classes that collaborate to fulfill a larger purpose. A micro-architecture is a set of such classes that collaborate to fulfill a larger purpose. From the point of view of reuse, they represent an attractive concept since they represent a higher level of reuse than class reuse. Reusing a micro-architecture implies reusing the design and implementation of a group of classes and their interactions, rather than just reusing the implementation of a single class. Due to central importance of groups of cooperating objects in OO systems it can be argued that a class, in general, represents

¹The term micro-architecture was introduced by Gamma in [25]

too fine grained a construct to adequately serve the purposes of reusable object-oriented component [64]. Interobject interactions between the classes in the micro-architecture must be fully understood before a micro-architecture can be reused.

Using a framework is typically composed of two activities [64]:

- defining any new classes that are needed as subclasses of existing classes in the framework
- configuring a set of objects by providing parameters to each object and connecting them.

It is clear that reusing a framework would necessarily involve the reuse of one or more micro-architectures in the framework. Further, reuse of frameworks introduces new complications not associated with the reuse of micro-architectures. To obtain a given behavior it is necessary to determine the answer to the following questions:

- From which existing classes should new classes be derived?
- Which methods should be overridden in those new classes?
- Which new objects should be created?
- How should objects be initially interconnected?

For effective reuse of frameworks, it is necessary to be able to answer such questions with minimal effort without having to inspect the source code. A summary of the state-of-the-art design pattern approaches in [50] shows that existing approaches for documentation and specification of design patterns, framework cookbooks and micro-architectures provide for most part an informal textual notation, graphic notation or use programming language notations. Current OO programming languages provide no support for specification and abstraction of interaction between objects; they only provide syntactic mechanism for implementing them. The existence of inter-object behavior in the system, and in particular the *behavioral dependencies* [30] which they imply cannot be easily inferred.

For, the same reasons discussed earlier, it is desirable to use formal specifications, rather than informal specifications or the implementation code itself, to provide formal representation for frameworks and describe the inter-object behavior within micro-architectures. It is also important that the interactions be described in a black-box manner, for the same reasons as it is necessary to describe class behavior in a

black-box manner. The work presented in [30] shows how this can be accomplished. *Contracts* are introduced to specify inter-object behavior by specifying behavioral compositions and obligations on participating objects. *Conformance declarations* are used to specify how specific classes, and thus their instances support the role and obligations of participants in the contract. Further, *Contracts* provide constructs for the refinement and inclusion of behavior defined in other contracts. *Refinement* allows for the specialization of contractual obligations and invariants of other contracts. *Inclusion* allows contracts to be composed from similar *contracts*. These constructs provide two distinct means to specify complex behavioral compositions in terms of simpler ones.

Nevertheless *contracts* have several limitations too. No formal syntax and semantics has been given for the specification language used to specify *contracts*. The work does not consider the specification of the behavior of individual classes, but only class interactions. *Contracts* have been found to be difficult to apply in certain situations especially specifying subclass relationships [50]. The abstraction level of the formal notation seems to be too close to OO programming languages.

In this thesis we address the issue of specifying inter-object interactions within a micro-architecture. Our approach identifies the issues involved in such collaborations and provides a three-tiered specification framework. The language we propose is an extension of Larch/C++

2.3.3 Design Pattern Reuse

Design patterns are employed by two kinds of designers. Framework designers use patterns as an aid in the construction of frameworks. They create frameworks for other designers to use and can be viewed as *suppliers* of design patterns and frameworks. The software designer can be viewed as a *user* of design patterns and frameworks. Framework users identify a general solution design pattern and use the appropriate micro-architecture to solve a specific design problem. Frameworks are thus designed for reuse and with reuse.

Employing design patterns within software reuse poses two fundamental problems [66]:

- how to produce design patterns with maximum potential for reuse – design for reuse

- how to design new systems making the most cost effective use of such components – design with reuse

Recent research at many industries have focussed on mining object-oriented systems for design patterns. Design patterns are discovered by mining existing applications and frameworks. However, *finding* patterns is much easier than *describing* them [25]. It is clear that from the point of view of reuse, describing patterns is very important, for in the absence of proper documentation pattern reuse is not possible. Several documentation styles have been proposed for describing patterns. Notable among them are GOF style [25], [54], AG Communication Systems HTML template, Peter Coad's style [14] and scores of other documentation styles found in [56].

Many pattern writers follow the paradigm of satiating their intellectual urges akin to poets composing poems. This poses the danger of reducing patterns to literary pieces than software components that can be reduced. It is clear that standardizing documentation styles for pattern descriptions is a mandatory first step to promoting their large scale reuse as well as for facilitating formal approaches to their documentation. Object-oriented software community is also divided on the abstraction level of design patterns. The patterns presented by Peter Coad [14] seem to be analysis level pattern, patterns in [25] are at a detailed design level patterns which use low level features available in C++ and SmallTalk. *Coding patterns* solve specific tasks [50] effectively in the realm of particular object-oriented programming language. SmallTalk and C++ coding patterns are quite popular in the industry. The patterns presented in Gamma et al. [25] are in the spirit of matured object-oriented designs and we shall focus on these type of patterns in this thesis.

Currently, the documentation styles for design patterns consist of informal textual notation, and/or graphic notation. Besides, the reasons mentioned in earlier sections for using formal specifications rather than informal specifications we believe that formal language for pattern description is necessary due to the following reasons:

- Case studies in industries have revealed [55], that developers encounter great amount of difficulty in understanding and implementing design patterns when they were described using only object diagrams and structured prose. This is due to the inherent limitations of the informal language, and lack of formal semantics for notations such as OMT and Object Collaboration Graphs.
- Design patterns tend to be difficult to understand in isolation due to lack of

detail in their high-level descriptions. This high level of description is however intentional and indeed is required to ensure wide applicability. Informal language is inadequate to document and communicate abstractions. There is more than one layer of abstraction encountered in design pattern descriptions and such abstractions can be captured precisely only by the use of formal methods.

- Patterns explicitly capture design knowledge that experienced developers only understand implicitly. For example, many developers intuitively understand the forces that cause certain solutions to be preferred over alternatives. However, these non-functional forces are not adequately captured by existing design methods and notations. A formal approach can aid in the documentation of such non-functional forces, design decisions and trade-offs and also help evaluate the design.
- Successful patterns capture both structure and behavior. Patterns that do not clearly describe dynamic behavior are difficult to understand and apply. One goal of formalism is to precisely specify these class and object relationships, and behavioral dependencies between objects in the pattern.
- Although each pattern addresses a particular, isolated design issue, in reality it is always part of a larger structure. So, a pattern may contain, be contained in, or be interlocked with other patterns, and will also interact with them [55]. Framework users typically use one or more patterns in conjunction in their architecture. Designing a formal language is a pre-requisite to formalizing the semantics of different kinds of relationships between patterns. These can further help in studying the concepts of *completeness* of a set of patterns.
- Johnson [34] introduced an informal *pattern language* that can be used for documenting a framework using the design patterns in the framework. Formal documentations of design patterns coupled with specification of inter-object collaborations within the corresponding micro-architecture in a framework can significantly lower the learning curve associated with frameworks and contribute to their reuse.

In addition to the above mentioned advantages, we believe that the most important motivating factor for ushering in formal methods is to reduce the *design pattern*

understanding time. Case studies in industries have revealed that the learning curve associated with design patterns is quite steep [25, 55]. Some studies suggest that *program understanding time* may be the dominant time in the entire software life cycle and thus the dominant cost [59]. In view of this, it might be construed that software reuse will be effective in increasing productivity only so far as it will be effective in reducing or eliminating the need for designers to read through verbose and voluminous descriptions of design patterns which they reuse.

Having thus motivated the need for a formal approach to documenting design patterns, we caution that such approaches will have certain limitations. These issues are dealt with in detail in Chapter 4. In this thesis we study the applicability of formal approaches to documenting design patterns and also provide outlines of a language that can be used to formalize certain aspects of design patterns that are essential to their reuse. It must be noted, that currently no formal documentation approaches exist.

2.3.4 Pattern Repositories

Besides the need for formal documentations, there is a need felt in the software community to create libraries of reusable components that reusers can use to retrieve information from such components. The Aesop system [49] developed at Carnegie Mellon University is an experimental platform that minimizes the cost of building systems by providing a generic infrastructure of common tools (design database, GUI, editors, protocol consistency checkers, Software Shelf etc.). This system uses a repository of design elements called Software Shelf that support the classification, storage and retrieval of architectural elements. The ITHACA project is an Espirit II project involving a number of European companies, research organizations, and universities to design and build an integrated application development and support environment based on object-oriented programming approach. However, not many details are available on the status of this project.

To promote large scale reuse of design patterns, the software community must be provided with a design pattern repository. This would enable designer to retrieve information from the pattern repository which can significantly lower the time involved in picking the right pattern for reuse and understanding it. In this thesis we address this problem and propose a conceptual schema of such a repository. The motivation

and the problem are discussed in detail in Chapter 5.

2.3.5 Other Levels of Reuse

Reuse-in-the-medium besides involving the reuse of such micro-architectures, involves an additional level of reuse, which comprises of the interaction of micro-architectures. At this level, the objects of reuse are no longer classes, but are themselves micro-architectures. This is in fact the reuse of system architectures (framework applications) that have been instantiated from the underlying framework.

The highest level of reuse, *reuse-in-the-large*, is the reuse of *application* objects, which are themselves independent systems and which are reused as they are, without being modified or extended in any way. We do not address these issues in this thesis.

Chapter 3

Specifying C++ classes in Larch/C++

Reuse in the small is concerned with the reuse of classes, methods and code fragments. The benefits of this reuse are maximized if classes are reused in a black-box fashion. In this chapter we motivate the choice of a specification language for specifying C++ classes to promote black-box reuse. We give a brief review of Larch/C++ and show the interface specification of a C++ class chosen from the Rogue Wave Library.

C++ has become one of the most popular object oriented languages used in industries. Several software development groups reuse class libraries, such as the Rogue Wave Tools.h++ and the Microsoft Foundation class library, in developing application software. For error free reuse the classes in the reuse library must be shown to be complete and correct. In our research [1, 2] we have applied the completeness criteria to many of these classes. In doing so we have written the interface specification of the classes in the formal specification language Larch/C++. In this section we discuss the features of Larch/C++ that led us to choose it over other specification languages for specifying C++ class interfaces.

Larch is a family of formal specification languages geared towards the specification of the observable effects of program modules, particularly modules which implement abstract data types. It provides a two-tiered approach to specification:

- In one tier, a Larch Interface Language (LIL) is used to describe the semantics of a program module written in a particular programming language. LIL specifications provide the information needed to understand and use a module

- interface. LIL doesn't refer to a single specification language but to a family of specification languages. Each specification language in the LIL family is designed for a specific programming language. The LIL for C++ is called Larch/C++. LIL specifications specify the interface of a procedure and its behavior. To formally specify the behavior of a function, Larch provides some clauses that assist the specifier to describe the states that the function is defined, what the function is allowed to change (modifies clause), the restrictions on the states and the arguments with which the client is allowed to call the function (requires clause), and the constraints on the function's behavior, when the function is called properly, which relate the pre-state and the post-state of the function (ensures clause). The pre-condition, expressed in the requires clause, and the post-condition, expressed in the ensures clause, are expressed
- in predicate logic, using logical assertions that contain terms of which formal meaning is specified in the other tier of Larch (LSL tier).
 - In the other tier, the Larch Shared Language (LSL) is used to specify state-independent, mathematical abstractions which can be referred to in LIL specifications. LSL is programming language independent and is shared by all LILs. These abstractions are called traits and are written in the style of an equational algebraic specification using the Larch Shared Language (LSL). Strictly speaking, Larch is a definitional language with equational axioms in the LSL-tier and Hoare-style axiomatic specifications in the interface tier. These specifications, written in LSL, are programming language independent and they can be used by any member of the LIL family. The abstractions defined by traits play the same role as the abstract models (e.g. sets, maps) in Z and VDM. From this point of view, the interface tier of Larch is very similar to a model-oriented specification such as VDM. This is a desirable property, as experience in the formal methods community suggests that model-oriented specifications are easier to read and write than other types of specifications.

Larch's two-tiered approach makes it possible to express module properties which are programming language dependent using a syntax and semantics which reflects the underlying programming language. This is achieved by providing constructs for expressing module properties such as parameter passing, side effects, exceptions, and concurrency using the syntax and semantics of the underlying programming language.

The syntax and semantics of each LIL also takes into account the syntax and semantics of the type system and the memory model of the underlying programming language.

For example, the syntax and semantics of C pointers, arrays, structs and other basic data types are built into Larch/C++. Similarly, C operators such as '*' and '→' are also built in. This built-in support facilitates the task of writing formal module specifications. Formal parameters in the formal specification can be referenced using the same syntax and semantics as in the underlying language.

In contrast to Larch, other specification languages such as Z and VDM, do not offer many of the capabilities described above. The result of this is that Larch module interface specification descriptions have the potential of being shorter than specifications written in other languages. In addition, they can also be clearer and more natural to developers who are accustomed to the syntax and semantics of the underlying programming language. Larch's two-tiered approach makes it as intuitive and expressive as other model-oriented specification languages such as Z and VDM.

In addition to Larch/C++, a few other languages have been proposed for the formal specification of object-oriented program modules. These include A++ [12, 13], Anna [45], Fresco [62] and the assertion language in Eiffel [48]. Larch/C++ differs from these languages in two important respects: (i) It provides built-in support for semantic constructs which are specific to C++ (e.g. pointers, built-in types, memory model) (ii) It provides the ability to write implementation independent specifications. Larch/C++ interface specifications can be implementation independent because they do not need to refer to the instance variables of a class to specify the semantics of its interface. Instead, the interface specifications refer to abstract sorts whose properties are defined axiomatically in traits.

Eiffel's assertion language and A++ also provide the ability to write specifications which do not refer to the instance variables of the class. Such specifications must refer only to the class's operations, permitting an algebraic-like specification style. Because specifications written in this style lack the ability to use abstract model variables, they are not nearly as expressive or intuitive as specifications written in Larch's two-tiered model-oriented style. In the case of Eiffel, the situation is even worse since the assertion language does not make it possible to express universally or existentially qualified assertions. Fresco methodology can be used to write implementation independent specifications of the behavior of software modules. However, Fresco is

geared towards the specification of Smalltalk classes. It does not provide any built-in support for C++ semantic constructs like Larch/C++ does.

In view of the above considerations, Larch/C++ is the most suitable specification language for specifying the behavior of C++ class interfaces for software reuse. In this chapter we shall present an overview of Larch/C++, summarize the recent additions and improvements to Larch/C++ specification language and also present our work in using Larch/C++ to specify and evaluate the black-box behavior of the C++ classes from the Rogue Wave Library Tools.h++.

3.1 Larch/C++ - A Quick Overview

Larch/C++ is a Larch-style interface specification language tailored to the C++ programming language. Its main objective is the formal specification of C++ program modules.

3.1.1 Larch Shared Language - LSL

The unit of specification in LSL is the trait. A trait contains a set of operator declarations, or signatures, which follows the **introduces** keyword, and a set of equational axioms, which follows the **asserts** keyword. A signature consists of operators whose domain and range are represented by *sorts*. An equational axiom specifies a set of constraints on the defined operators.

The semantics of LSL traits is based on multisorted first-order logic with equality rather than on an initial, final or loose algebra semantics used by other specification languages [7, 22, 27, 53]. Each trait denotes a *theory*¹ in multisorted first-order logic with equality. The theory contains each of that trait's equations, the conventional axioms of first-order logic with equality, everything which follows from them and nothing else. This means that the formulas in the theory follow only from the presence of assertions in the trait – never from their absence.

A trait definition need not correspond to an abstract data type (ADT) definition since an LSL trait can define any arbitrary theory of multisorted first-order equational logic. For example, a trait can be used to define theory of mathematical abstractions such as equivalence relations, which do not correspond to abstract data types. For

¹A *theory* is a set of logic formulas having no free variables.

LSL traits that define an ADT, there is a sort referred to as the *distinguished sort*, sometimes also called the *principal sort* or *data sort*.

The semantics of $=$ and $==$ in LSL are exactly the same as in conventional algebraic specification languages, except that the operator $=$ binds more tightly than does the operator $==$. LSL traits can be augmented with checkable redundancies in order to verify whether intended consequences actually follow from the axioms of the trait. The checkable redundancies are specified in the form of assertions that are included in the *implies* clause of a trait and can be verified using LP [29].

LSL also provides a way of putting traits together, one of which is through an *includes* clause. A trait that includes another trait is textually expanded to contain all operator declarations, **generated by** clauses, and axioms of the included trait. Boolean operators (true, false, not, \vee , \wedge , \rightarrow , and \leftrightarrow) as well as some heavily overloaded operators (if-then-else, $=$) are built into the language; that is, traits defining these operators are implicitly included in every trait.

The theory of a trait can also be strengthened by adding a **generated by** or a **partitioned by** clause. The **generated by** clause states the operator symbols that can generate all values of a sort. The **partitioned by** clause provides additional equivalences between terms. It states that two terms are equal if they cannot be distinguished by any of the functions listed in the clause.

3.1.2 Larch/C++

The Larch/C++ specification of a C++ function specifies not only the behavior of the function, but exactly how that function is called from the C++ code. The details of how to call a C++ function, the name, return type, and argument types, are called are part of the specification of the interface function.

Functions are specified in Larch/C++ using Hoare-style pre- and post-conditions. The header of a function specification is the same as that of a C++ function definition. The body describes the effect of function invocation using a pair of predicates following the keywords **requires** and **ensures**. The predicate following **requires** is a pre-condition that must be satisfied to invoke the specified function. The predicate following **ensures** is a post-condition that the specified function establishes upon termination. The semantics of function specification is that the pre-condition of the

state transformation must logically imply the post-condition of the state transformation. For functions that change the values of the objects, the body of the function specification must include the **modifies** clause. Only objects listed in the **modifies** clause are allowed to change their values as the result of function invocation. In a function that mutates an object or a variable, there are two different values for the same object; the value in the pre-state and the one in the post-state. The value of the object in the pre-state is denoted by a hat-ed (^) identifier, while the post-state value is represented by a primed (') identifier. If neither of (^) nor (') is used with the object name then the object itself is considered as a memory location and not the object value.

The syntax for data members and member functions in interface specifications are almost the same as in a C++ program. The Larch/C++ reserved word *this* (or *self*) is used in member function specifications and means the same thing as the C++ reserved word *this*, a pointer to the object of the specified class. The Larch/C++ reserved word *self* is a shorthand for $*(this \setminus any)$. The suffix *any* is like (') or (^), and extracts the value of *this* in some visible state. As in C++, Larch/C++ member functions can be public, protected, and private.

The trait **SetTrait** defines the terms used to denote the abstract values of the set as well as the mathematical properties of the set. The **SetTrait** defines the distinguished sort *C* to denote the abstract values of the Set. The **generated by** clause indicates that all the abstract values of the sort Set are generated by operator symbols $\{\}$ and *insert*. The **partitioned by** clause indicates that the operator symbol \in partitions the abstract values of Set objects into equivalence classes of LSL terms of sort *C*.

LSL specification for the Set

SetTrait(*E*, *C*) : **trait**

 % Essential finite-set operators

introduces

$\{\}$: $\rightarrow C$

insert : $E, C \rightarrow C$

delete : $E, C \rightarrow C$

$- \in -$: $E, C \rightarrow Bool$

$isEmpty : C \rightarrow Bool$

asserts

C generated by $\{\}$, $insert$

C partitioned by \in

$\forall s : C, e, e_1, e_2 : E$

$\neg(e \in \{\})$

$e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$

$isEmpty(\{\})$

$\neg isEmpty(insert(e, s))$

$e \in s \Rightarrow \neg isEmpty(s)$

$delete(e, \{\}) == \{\}$

$delete(e_1, insert(e_2, s)) == \text{if } e_1 = e_2 \text{ then } s \text{ else}$

$insert(e_2, delete(e_1, s))$

In the Larch/C++ specification for the class Set, the **uses** clause indicates that the Larch/C++ interface is expressed with the vocabulary of the LSL trait SetTrait. All the terms in the pre- and post- conditions of the function specifications come from this trait. The type-to-sort mapping, which is given between the parenthesis following the names of the used trait, says that the abstract values of the C++ Set objects are specified to be those of the LSL sort C in SetTrait. (In LSL a sort is the type of an LSL term; the word type is used only to refer to C++ types). The type to sort mapping makes the connection between the C++ world and the LSL (mathematical) world.

Larch/C++ specification for the Set

class Set

{ uses SetTrait(IntSet for C, int for E);

public:

IntSet()

{

modifies *self*;

ensures $self' = empty$;

```

    }

~IntSet()
{
    modifies self;
    ensures trashed(self);
}

IntSet& insert(int i)
{
    modifies self;
    ensures  $self' = insert(self^, i) \wedge result = self$ ;
}

IntSet& delete (int i)
{
    modifies self;
    ensures  $delete(self^, i) = self' \wedge result = self$ ;
}

Bool isEmpty()
{
    ensures if isEmpty(self^) then result = TRUE
    else result = FALSE;
}

};

```

The Larch/C++ specification for the Set specifies a constructor, a destructor, and three public member functions: *insert*, *delete*, and *isEmpty*. The destructor uses the Larch/C++ reserved word **trashed** (whose semantics is provided in the LSL layer) to state that the object **self** is no longer available. The terms *insert*, *isEmpty* and *delete* which appear in the pre- and post-conditions refer to the LSL operators, not to the member functions having the same name. All C++ declarations are legal in Larch/C++ interface specifications; for example, member functions can be virtual,

static, friend or inline; they all have their C++ meaning. The Larch/C++ keyword **result** can only be used in post-conditions and it denotes the function return value. The sort of **result** is the sort associated with the return type specified for the function. For example, in the interface specification for the class **Set** the member function **isEmpty** returns sort **Bool**.

3.2 Improvements on Larch/C++

Larch/C++ is still in the process of development. Not all features have been implemented. Currently only the parser is available, some parts of which are still in the development stage. Many versions and upgrades have been created with numerous changes in the syntax and the semantics of the language. The Larch/C++ manual [38] is a very fast changing document describing the language as the changes take place. In this section a summary of the most important new features is given. These new features have been added recently to the language and are extensively analyzed in the Larch/C++ reference manual [38].

3.2.1 Composite Sorts

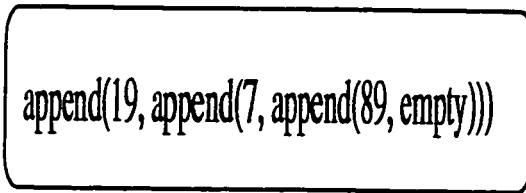
Composite sort names have been added only recently to Larch/C++. In the absence of any documentation, several users of the Larch/C++ community were consulted. The consensus is that composite sort names are to be treated exactly as atomic sort names except that when it comes to renaming. The rule is that the rename binding is propagated through the components. For example, if $Loc[E]$ is a composite sort and E is an atomic sort, with the renaming E to F , the bindings

$$\begin{aligned} e &: \rightarrow E \\ f &: \rightarrow Loc[E] \end{aligned}$$

become

$$\begin{aligned} e &: \rightarrow F \\ f &: \rightarrow Loc[F] \end{aligned}$$

Composite sorts are not parameterized sorts: ‘ $Loc[E]$ ’ is a sort name that has ‘ Loc ’



```
My_list: append(19, append(7, append(89, empty)))
```

Figure 1: Larch/C++ Model of a List Object

and ‘ E ’ as components; ‘ E ’ does not have any special status. In particular E is not a parameter.

Composite sort names are used in order to be able to distinguish between T values and T objects. In the past, an object of type T was denoted by Obj_T . One problem with this approach is that while renaming T , the specifier had to rename Obj_T . With the composite sort names, as shown in the example above, the renaming of the object sort ($Loc[E]$ in the example) is done implicitly. Also, Obj_T did not have any fixed semantics. The specifier had to explicitly specify the semantics of an object in an LSL trait. The new version of Larch/C++ has provided a few LSL traits that generally specify the behavior of objects (mutable, immutable, typed and untyped), and the semantics of states [38].

3.2.2 The Formal Model of Objects, Values, and States

According to the designers of C++, “an object is a region of storage.” All objects have an address, or location. They store values, which in the context of a specification are called *abstract values*. Figure 1 illustrates the concept of objects and values by showing the Larch/C++ model of a list object in a certain state. A state associates each object with an abstract value. In this example the object *My_List* is associated with the abstract value *append(19, append(7, append(89, empty)))*. Objects in Larch/C++ are modelled using several traits. The main trait is *TypedObj*, which handles the translation between typed objects and values and the untyped objects and values used in the trait *State*. Objects can be either mutable or constant (immutable). Mutable objects are modelled by sorts with names of the form $Obj[T]$, which is the sort of an object containing an abstract value of sort T .

The trait *MutableObj* gives the formal model of mutable objects by adding the capability of mutation to the trait *TypedObj*. Constant objects are modelled by

sorts with names of the form *ConstObj[T]*, which is the sort of a constant object containing abstract values of sort *T*. The trait `ConstObj` gives the formal model of constant objects. A state is a mapping from objects to values. During the course of execution, a program creates objects and binds values to objects. A state captures the set of objects that exist at a particular time and their bindings. The trait `State` gives the formal model of states used by Larch/C++. It defines the state (sort `State`) as a mapping between untyped objects (sort `Obj`) and abstract values (sort `Val`).

3.2.3 Declarations and Declarators

C++ provides numerous kinds of declarators for every possible declaration. Larch/C++ has incorporated these declarators both in syntax and semantics. In Larch/C++, a declaration is needed to specify that the C++ module that implements the specification must have the same declaration, and to give information about the declared construct's type, and other attributes. Although Larch/C++ syntax for the declarators tries to be identical to the C++ one, there are some minor differences which were created purposely in Larch/C++ in order to resolve some ambiguities in the C++ grammar.

In a declaration, a declarator defines a single object, function or type, along with its name. The semantics of each declarator is defined using LSL traits which are built in Larch/C++. For instance when declaring an integer global variable, Larch/C++ implicitly uses the `int` LSL trait. Using the following operators a declarator may refine an object's type : `*` (pointer), `::*` (pointer to member), `&` (reference), `[]` (array), `()` (function). A variable declared globally, or a formal parameter passed to function using one of these declarators, has a particular sort. Pointers have the `Ptr` sort generator as part of the sort name of the term. References use the `Obj` sort generator, and arrays use the `Arr` sort generator. `ConstObj[cpp_function]` is the sort of a non-member function and `ConstObj[cpp_member_function]` is the sort of a member function. The semantics of these sorts are described using the LSL traits discussed previously. In Tables 1 and 2 there is a summary of the sorts that global variables, and formal parameters take when declared with the various declarators:

Tables 1, 2 assume that `IntList` is a structure of the following type:

```

struct IntList{
int val;
IntList *next;
}

```

In these tables a term x of sort $Ptr[Obj[T]]$ is a pointer that points to an object that contains an abstract value of sort T . To obtain the object that the pointer points to, the operator $*$ must be used. Therefore, $*x$ would be of sort $Obj[T]$. A term x of sort $Arr[Obj[T]]$ is an array of objects that contain abstract values of sort T . To obtain any of these objects, the operator $[]$ and the integer index of the particular object are used. A *structure* or a *union* declared globally is an object. Since C++ parameters are passed by value (except for reference parameters), a *structure* or a *union* passed as a parameter to a function is not an object but simply a tuple of the respective fields. That is the reason why in the first table the sort of the global variable of type *IntList* is $ConstObj[IntList]$ and in the second table the sort of the formal parameter of type *IntList* is $Val[IntList]$.

3.2.4 State Functions

An object can be in an infinite number of states through out its entire life. Not all states are visible to the client of a class interface. In particular, only a very limited number of states are visible. The states that are not visible to a client are called *internal object states*. Therefore, the states that are of particular interest to the class interface are:

- the *pre-state*, which maps objects to their values just before the function body is run, but after parameter passing,
- the *post-state*, which maps objects to their values at the point of returning from the call (or signalling an exception), but before the function parameters go out of scope.

To obtain an object's abstract value in a particular state (provided that the object is assigned in that state), a *state function* must be used. There are four state functions in Larch/C++ :

- $\backslash pre$ or $\hat{\ } :$ it obtains the abstract value of an object in the pre-state,

- `\post` or `'` : it obtains the abstract value of an object in the post-state,
- `\any` : it obtains the abstract value of an object in no particular state. This state function is usually used when the object is immutable (not modifiable) and therefore, its abstract value is the same in both pre-state and post-state.
- `\obj` : it is used to explicitly refer to an object itself, instead of its abstract value. It is only used for emphasis.

The state functions can only be applied to terms that denote objects and their sort is either $Obj[T]$ or $ConstObj[T]$ for some type T . The sort of any object of type T that has been applied one of the first three state functions (`\pre`, `\post`, `\any`) is the same as that of the object's but without the leading Obj or $ConstObj$ sort generator. When the `\any` state function is applied to an object, the sort of the term with the state function is the same as the sort of the object. For example, if the sort of x is $Obj[int]$ then the sort of x' is int and the sort of $x\any$ is $Obj[int]$.

3.2.5 New features in Function Specifications

Several new clauses have been added lately to the syntax and semantics of Larch/C++.

- The **constructs** clause is an equivalent of the modifies clause. Larch/C++ provides this clause for the added convenience of the reader/specifier. This clause is used in constructor functions in order to express that an object is not only modified but there is memory allocated for it, and its attributes are initialized.
- The **trashes** clause is used for any function that trashes objects. In Larch/C++ the trashing of an object is done whenever the object was assigned in the pre-state and not assigned in the post-state, or when the object was allocated in the pre-state and not allocated in the post-state. The trashes clause lists a set of objects that may be trashed from the function.
- The **claims** clause, as in LCL, contains a predicate which does not affect the meaning of a function specification, but rather describes redundant properties which can be checked by a theorem prover. The following example illustrates the use of the trashes and claims clauses.

```

void dec_ref(char *cp, int & ref_count)
{
  requires allocated(cp, pre) ∧ assigned(ref_count, pre) ∧
           ref_count^ >= 1;
  modifies ref_count;
  trashes *cp;
  ensures ref_count' = ref_count^ - 1 ∧
           (if ref_count' = 0 then trashed(*cp)
            else ¬isTrashed(*cp, pre, post));
  claims ref_count' > 0 ⇒ ¬isTrashed(*cp, pre, post);
}

```

- The **let** clause can appear in any function specification. It can be used in order to abbreviate expressions that will be used many times in the function specification (requires, ensures, example clauses). The following example illustrates the use of this clause.

```

imports BankAccount;
void transfer(BankAccount& source, BankAccount& sink, long int cts)
{
  let amt:Q be dollars(cts),
      presrc:Q be source^, presink:Q be sink^,
      oldsrc:Q be presrc.credit, oldsink:Q be presink.credit;
  requires source! = sink ∧ assigned(source, pre) ∧ assigned(sink, pre)
           ∧ oldsrc >= amt ∧ amt >= 0;
  modifies source, sink;
  ensures sink' = set_credit(presink, oldsink + amt)
           ∧ source' = set_credit(presrc, oldsrc - amt);
}

```


- The **example clause** can be used to give the reader/specifier a concrete example of the function behavior. Examples do not change the meaning of a specification.

The new keywords that have been recently added in the language are the following:

- The keyword **allocated** can be used in a predicate (requires, ensures clauses) in order to specify that an object is allocated at a certain state. An object can exist without being allocated.
- The keyword **assigned** can be used in a predicate (requires, ensures clauses) in order to specify that an object has a well-defined value [15]. In other words an object is assigned if its value is initialized.
- The keyword **fresh** can only appear within an ensures clause predicate and it is used to specify that an object was not allocated in the pre-state, and it is allocated in the post-state. The following example illustrates the use of fresh in function specifications.

```

typedef int *ratl;
ratl make_ratl(int n, int d)
{
    requires  $d > 0$ ;
    ensures  $assigned(result, post) \wedge size(locs(result)) = 2$ 
            $\wedge (result[0])' = n \wedge (result[1])' = d$ 
            $\wedge fresh(result[0], result[1]);$ 
}

```

- The keyword **unchanged** is used within predicates when there is the need to express that a modifiable object is not modified.
- The keyword **reach** can be used with an object to denote the set of all objects reachable from that object.

- The keyword **liberally** can be used in an ensures or claims clause when the predicate gives a *partial-correctness* specification. In any *partial-correctness* or *total-correctness* specification, if the pre-condition is true, and if the function terminates normally, then the post-condition must be true. In *partial-correctness* specifications normal termination is not guaranteed, even when the pre-condition is true. Therefore a specification that does not use the keyword *liberally* is a *total correctness* specification and a specification that uses the keyword *liberally* is a *partial-correctness* specification.

Following are some more new constructs recently built in Larch/C++.

- In C++, a **default value** can be given for a formal argument. This means that when calling the particular function without supplying a value for the particular formal argument, then that argument takes the specified default value. This is also the case in Larch/C++. The syntax is identical.
- Many times functions are specified better in several different cases. Larch/C++ provides the specifier with the ability to specify a function using many cases. For every case, the specifier is able to specify a different requires, modifies, trashes, ensures, let, and claims clause. The predicates in the requires clauses should be exclusively disjoint.
- In C++, the interface of a function can declare what **exceptions** the function can throw. The same is also true in Larch/C++ which specifies a function that throws an exception by considering its result to be either the normal result or an exception result. The following example illustrates the specification of exceptions, and the use of cases in a function specification.

```

imports Overflow;
void inc2(int& i) throw(Overflow)
{
    requires assigned(i, pre) ∧ î + 2 ≤ INT_MAX;
    modifies i;
    ensures result = theVoid ∧ i' = î + 2;
    requires assigned(i, pre) ∧ î + 2 > INT_MAX;

```

```

    ensures thrown(Overflow) = theException;
}

```

3.2.6 New Features in Class Specifications

- C++ implicitly defines several member functions for the user if they are not explicitly defined. These are a default constructor, a copy constructor, a destructor, and an assignment operator, provided that they are not explicitly declared. Larch/C++ implicitly provides an appropriate specification for these implicitly defined functions. These are called **implicit specifications**.
- Larch/C++ provides the **simulation clause** where the user may specify the relationship between super-type and sub-type. As will be seen in Chapter 7, the specification of the super-type might be expressed in a different mathematical domain. The use of the simulation function gives valid meaning to the super-type's specifications in the context of the sub-type. The behavior of the simulation function is specified using an LSL trait.
- Larch/C++ allows the specifier to specify **history constraints** on the values that an object may take. It specifies a relationship between each pair of visible states ordered in time. An object may remain immutable through its entire life. As an example, in a *Person* object with fields *Name* and *Age*, the *Age* field may only be increasing. These cases can be specified using the **history constraint clause**. A history constraint is syntactic sugar in Larch/C++. The same behavior can be specified in the predicate of every member and friend function's postcondition, except for the constructors.
- Larch/C++ provides an **invariant clause** which allows the user to specify an invariant property that must be true during the entire life time of an object of the particular specified class. There are two equivalent ways of thinking about invariants. The first is that the invariant is conjoined to each pre and postcondition of each member function in the specification. The second is that the invariant is true in all visible states. Within an implementation of a member function, an invariant may be temporally violated. This is acceptable, since any intermediate state of the class variable is invisible to any clients.

- Larch/C++ can also specify **friendship relationships**. A friendship specification records information that may be needed in the implementation phase of a class. As in C++, friendship grants access to the private interface of a class to particular functions or to all member functions in some class.

The following is a class interface specification that includes many of the features mentioned in this section.

```

class Person
{
  uses PersonTrait, cpp_string; // age interpreted as number of years old
  invariant  $len(self\any.name) > 0 \wedge self\any.age \geq 0$ ;
  constraint  $self^.age \leq self'.age$ ; // age can only increase
  public:
  Person(const char *moniker, int years)
  {
    requires  $nullTerminated(moniker, pre) \wedge lengthToNull(moniker, pre) > 0$ 
       $\wedge years \geq 0$ ;
    modifies self;
    ensures  $self'.name = uptoNull(moniker, pre) \wedge self'.age = years$ ;
  }
  virtual ~Person()
  {
    ensures true;
  }
  virtual void change_name(const char *moniker)
  {
    requires  $nullTerminated(moniker, pre) \wedge lengthToNull(moniker, pre) > 0$ ;
    modifies self;
    ensures  $self'.name = uptoNull(moniker, pre)$ 
       $\wedge self'.age = self^.age$ ;
  }
  virtual char * name() const

```

```

    {
        ensures nullTerminated(result, post) ∧ fresh(objectsToNull(result, post))
            ∧ uptoNull(result, post) = self\any.name;
    }
virtual make_year_older()
{
    requires self^.age < INT_MAX;
    modifies self;
    ensures self' = set_age(self^, self^.age + 1);
    example self^.age = 29 ∧ self'.age = 30;
    claims self'.name = self^.name;
}
- virtual int years_old() const
{
    ensures result = self\any.age;
}
};

```

3.3 Motivation for using Rogue Wave Software

The Rogue Wave Tools.h++ [51] is a rich, robust and versatile C++ foundation class library. Virtually any programming chore can be done using classes from the library. Tools.h++ is an industry standard. The Rogue Wave Tools.h++ library is used by developers in several industries. Moreover, the library classes are well structured, well documented and are usable in isolation.

Tools.h++ consists mostly of a large and rich set of concrete classes that are usable in isolation and do not depend on other classes for their implementation or semantics. The concrete classes consist of a set of simple classes (such as date, time, string), and three different families of collection classes (collection classes based on templates, collection classes that use preprocessor <generic.h> facilities, "Smalltalk-like" classes for heterogeneous collection). The library also includes a set of abstract data types

(ADTs), and corresponding specializing classes that provide a framework for persistence, localization, and other issues. All collection classes have a corresponding iterator.

It is claimed [51] that the Rogue Wave Tools.h++ class library is built to achieve five basic goals: efficiency, simplicity, compactness, and predictability. This perfectly fits in with the criteria for black-box reuse. Because of these reasons, the Rogue Wave Library *tools.h++* was chosen for formal specification.

3.4 Interface Specifications of Rogue Wave Classes

There are two main goals in developing interface specifications for Tools.h++, the Rogue Wave Library of C++ classes: to identify and remove ambiguities in the informal descriptions [51] and evaluate the advantages and disadvantages of using Larch/C++ for specifying an industry standard tool set.

The Rogue Wave library contains more than 100 C++ classes and they have been ported to many operating systems, compilers, and application frameworks. These classes can be classified into simple classes, Smalltalk like collections, templates, and "generic" collection classes. The classes for which specifications were completed include simple classes **RWDate**, **RWTime**, **RWZone**, **RWFile**, **RWBTreeOnDisk**, **RWCSubString**, **RWCString**, **RWCRegexp**, **RWBag**, and **RWBinaryTree**, representatives of Smalltalk-like collection classes **RWCollectable**, **RWHashTable**, **RWSListCollectable**, **RWSequenceable**, **RWSListCollectablesIterator** and **RWOrdered**. The template classes for which we have written specifications include **RWTPtrVector**, **RWTPtrOrderedVector**, **RWTIsvDlist**, **RWTIsvDlistIterator**, and **RWTIsvSlistIterator**. All the specifications and their LSL traits are given in reports [1] [2]. They have been checked for their syntactic and semantic correctness. No representatives of generic collection classes was chosen, the rationale being that once a suitable mechanism for formal specifications for all required groups of classes, the remaining classes from the library can be specified by importing class specifications that we have completed and by reusing the LSL traits created by us in writing the specification of the above classes. For example, Figure 2 shows the subset of classes for which specifications have been created. In this figure $X \rightarrow Y$ means that X imports Y. Since imports relation is transitive, in order to

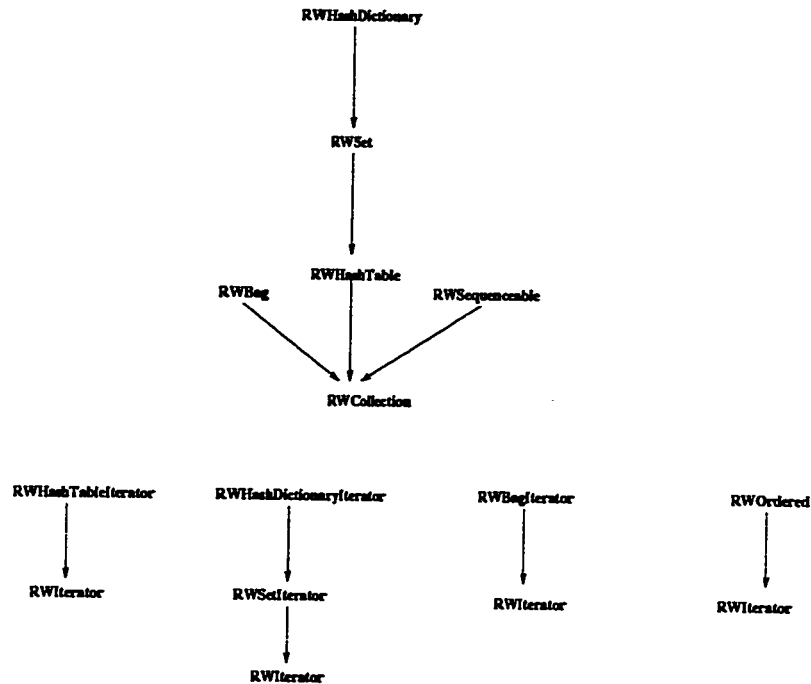


Figure 2: Import List Specification of a few classes

specify X , we need to create the specification of classes in the transitive closure of its import list. In particular, the specification of `RWDLISTCollectablesIterator` requires the specification of seven other classes in Figure 2. In addition, we also need to create the LSL traits necessary to capture the abstractions in these classes. By judiciously choosing the classes, we minimize redundancy and maximize reuse of LSL traits.

As part of this project I have developed interface specifications for Smalltalk-like collection classes `RWBag`, `RWBagIterator`, `RWHashTableIterator`, `RWHashDictionary`, `RWHashDictionaryIterator`, `RWOrdered` and `RWOrderedIterator`. Below the specification of `RWOrdered` is discussed.

3.4.1 Interface Specification of `RWOrdered`

We discuss in this section the interface specification of the class `RWOrdered`, which represents a group of ordered items accessible by an index (not necessarily an external key). The inheritance hierarchy for this class is that `RWOrdered` inherits from `RWSequenceable` which in turn from `RWCollectable`. The LSL traits created for supporting the abstractions of these classes are shown in Figure 3.

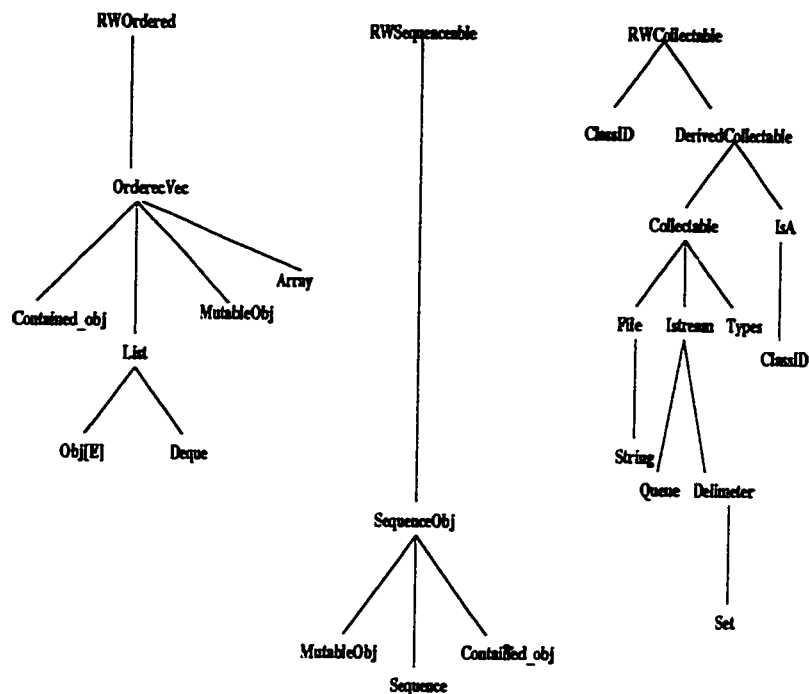


Figure 3: LSL Traits in the Specification of Ordered Vector

The ordering of elements in **RWOrdered** is determined by the order of insertion and removal of elements with duplicates allowed. An object stored by this class must inherit from the abstract base class **RWCollectable**. Class **RWOrdered** is implemented as a vector of pointers, allowing for more efficient traversing of the collection than the linked list classes in the library.

Class **RWOrdered** has one constructor and 29 member functions. The basic notion to be formalized here is that of “ordering” for any finite collection of objects. The ordering of elements in **RWOrdered** objects is determined by the order of insertion and removal of elements (with duplicates allowed). So, we wrote the LSL trait **OrderedVec**, which models such a collection. This trait reuses the **Array** and **List** traits from the LSL library and models the behavior of the collection as a finite sequence of elements with the given ordering. The library traits **MutableObj** and **contained_objects** specify respectively that the **OrderedVec** objects are mutable and each abstract value of **OrderedVec** object is in fact a container of objects. The traits **MutableObj** and **contained_objects** are reused from the interface specifications of **RWSequenceable** and **RWCollectable**. The specification of **RWOrdered** is imported by **RWOrderedIterator** and is inherited by **RWSortedVector**.

OrderVec - LSL Specification

OrderedVec(*S*, *Obj*[*E*]) : **trait**

includes *List*(*Obj*[*E*] for *E*, *S* for *C*), *Array*(*Obj*[*E*], *S*),
MutableObj(*E*), *contained_objects*(*S*)

introduces

nitem : *S* → *int*

isEmpty : *S* → *Bool*

NumEqualObjects : *S*, *Obj*[*E*], *State* → *int*

∈ : *Obj*[*E*], *S* → *Bool*

asserts

∀ *s* : *S*, *n* : *int*, *e*, *e*₁ : *Obj*[*E*], *st* : *State*

nitem(*create*(*n*)) == 0

nitem((*s* ⊢ *e*)) == *nitem*(*s*) + 1

isEmpty(*s*) == (*s* = *create*(*n*))

NumEqualObjects(*create*(*n*), *e*, *st*) == 0

NumEqualObjects((*create*(*n*) ⊢ *e*₁), *e*, *st*) == **if** ((*e*!*st*) = (*e*₁!*st*))

then 1 **else** 0

NumEqualObjects((*s* ⊢ *e*₁), *e*, *st*) == **if** ((*e*!*st*) = (*e*₁!*st*)) **then**

(*NumEqualObjects*(*s*, *e*, *st*) + 1)

else *NumEqualObjects*(*s*, *e*, *st*)

¬(*e* ∈ *create*(*n*))

(*e* ∈ (*s* ⊢ *e*))

RWOrdered - Larch/C++ Interface Specification

imports RWSequenceable;

class RWOrdered : public RWSequenceable

{

uses OrderedVec(RWCollectable for E, RWOrdered for S);

RWOrdered(size_t size = RWDEFAULT_CAPACITY)

{

```

        contracts self;
        ensures self' = create(size);
    }
RWBoolean operator==(const RWOrdered& od) const
{
    ensures result = ((nitem(self\any) = nitem(od\any)) ∧
        ∀i : int(i ≥ 0 ∧ i ≤
nitems ∧ NumEqualObjects(self\any, od[i], any)));
}
virtual RWCollectable* append(RWCollectable* a)
{
    modifies self;
    ensures if nitem(self^) < maxIndex(self^)
        then result = a ∧ self' = self^\postcat(a*)\any
        else result = NULL;
}
virtual RWCollectable*& at(size_t i) throw(int)
{
    requires i ≥ 0 ∧ i < nitem(self\any);
    ensures ((result')*)\post = self\any[i];
    requires i < 0 ∨ i ≥ nitem(self\any);
    ensures thrown(int) = RW BoundsErr;
}
virtual const RWCollectable* at(size_t i) const throw(int)
{
    requires i ≥ 0 ∧ i < nitem(self\any);
    ensures (result*)\post = self\any[i];
    requires i < 0 ∨ i ≥ nitem(self\any);
    ensures thrown(int) = RW BoundsErr;
}
virtual void clear()
{
    modifies self;
    ensures self' = empty;
}

```

```

    }
virtual size_t entries() const
{
    ensures result = nitem(self\any);
}
virtual RWCollectable* find(const RWCollectable* a) const
{
    requires ¬(a = NULL);
    ensures result* = (if(∃i : Int((self\any)[i]!any = (a*)\any ∧
        ∀j : Int(j < i ⇒ ¬((self\any)[j]!any = (a*)\any))))
        then (self\any)[i]
        else NULL);
}
virtual RWCollectable* first() const
{
    ensures if self\any = empty
        then result = NULL
        else result* = self\any[0];
}
virtual size_t index(const RWCollectable* a) const
{
    requires ¬(a = NULL);
    ensures result* = (if(∃i : Int((self\any)[i]!any = (a*)\any ∧
        ∀j : Int(j < i ⇒ ¬((self\any)[j]!any = (a*)\any))))
        then i
        else result = RW_NPOS);
}
virtual RWCollectable* insertAt(size_t indx, RWCollectable* e) throw(int)
{
    requires ¬(e = NULL) ∧ indx >= 0 ∧ indx <= nitem(self\any);
    modifies self;
    ensures nitem(self') = nitem(self) + 1 ∧ self'[i] = (e*)\any ∧
        ∀j : Int(j >= indx ∧ j <= nitem(self') ∧ self'[j + 1] =
self[j]) ∧ result = e;
    requires ¬(e = NULL) ∧ (indx < 0 ∨ indx >= nitems(self\any));
}

```

```

        ensures thrown(int) = RWBoundsErr;
    }
virtual RWBoolean isEmpty() const
    {
        ensures result = isEmpty(self\any);
    }
virtual RWCollectable* last() const
    {
        ensures if self\any = empty
            then result = NULL
            else (result*) = self\any[nitem(self\any) - 1];
    }
virtual size_t occurrencesOf(const RWCollectable* a) const
    {
        ensures result = NumEqualObjects(self\any, (*a), any);
    }
virtual RWCollectable* prepend(RWCollectable* a)
    {
        requires ¬(a = NULL);
        modifies self;
        ensures if nitem(self^)^ < maxIndex(self^)^
            then self' = (a*)\any\precatself^ ^ result = a
            else result = NULL;
    }
RWCollectable* remove(const RWCollectable* a)
    {
        modifies self;
        ensures if (∃i : Int((self\any)[i]!any = (a*)\any ^
            ∀j : Int(j < i ⇒ ¬((self\any)[j]!any = (a*)\any))))
            then (result* = self^[i] ^ self' = remove(self^, self^[i]))
            else (result = NULL ^ unchanged(self));
    }
virtual resize(size_t N)
    {

```

```

    modifies self;
    ensures if ( $N \geq nitem$ )
        then  $self' = create(N)$ 
        else  $unchanged(self)$ ;
    }
};

```

3.5 Experience in formalizing Rogue Wave library classes

Our experience in this project can be classified into two categories – the first kind arose while detecting and fixing the ambiguities and inconsistencies in the informal descriptions of tools.h++ [51]; the second kind arose while creating Larch specifications for reuse and with reuse. The project helped us to understand the usefulness and significance of using Larch/C++ in providing unambiguous specifications of black-box behavior of C++ modules (in contrast to the inherent ambiguities and inconsistencies found in the informal descriptions provided). Methodologies have also been developed to test and evaluate the specifications that were written. LSL provides the facility to specify abstractions by reusing existing abstractions and we exploited this feature to create reusable LSL traits and reuse previously specified traits.

3.5.1 Ambiguities and Inconsistencies

During the process of writing formal specifications for the selected classes it was necessary to write test programs in order to resolve the ambiguities and incompleteness in the informal descriptions. One of the most common ambiguity that we found in the informal description is in the description of functions that take a pointer as a parameter. It is required that the parameter is not NULL before the execution of the function. The informal description does not restrict the use of the function with a NULL parameter. This ambiguity in the informal specification resulted in the crashing of the test drivers written to test the class. A careful walk-through the code showed us the need there was need for a pre-condition in the function which stated that the pointer parameter should not be NULL before the execution of the function.

In some cases we specified in the post condition the kind of exception thrown. Some functions that have the above ambiguity in the class `RWOrdered` are:

- `virtual RWCollectable* find(const RWCollectable* a) const`
- `virtual size_t index(const RWCollectable* a) const`
- `virtual RWCollectable* insertAt(size_t indx, RWCollectable* e)`
- `virtual RWCollectable* prepend(RWCollectable* a)`

For more details on the ambiguities and inconsistencies found in the `RWOrdered` Class description we refer the reader to the reports [1, 2].

Another ambiguity is in the informal description for the copy constructor and the assignment operator. From the statement that a “shallow copy of the formal parameter is made”, it sounds as though both the copy and assignment have the same behavior. However, it is clear from an examination of the code that in the case of the copy constructor, the self object is created and a shallow copy of the formal parameter is made, while in the case of the assignment the self already exists and is assigned to the contents of the formal parameter. This is clear from the constructs clause of our instance specifications (see for instance `RWHashDictionary`, `RWBag`. Inheritance in Rogue Wave often means code redefinition and not strict subtype inheritance since we often find inherited functions being redefined to have a different behavior in the subclass than the behavior in the superclass. Our interface specifications are quite useful in such situations since it is easier to identify the inconsistency in the behavior of an interface function in the subclass and the superclass using a formal specification than an informal one. We also found ambiguities arising due to lack of clarity in describing state changes. For example, the informal description of the remove operation in classes `RWHashDictionaryIterator` and `RWHashTableIterator` merely states that the item at the current iterator position is removed from the collection; but fails to mention the state of the collection after the removal. However, according to the implementation of this class, after the removal of an item the iterator points to the position that immediately precedes the position of the removed item. This behavior is specified in the `Iterator` LSL trait.

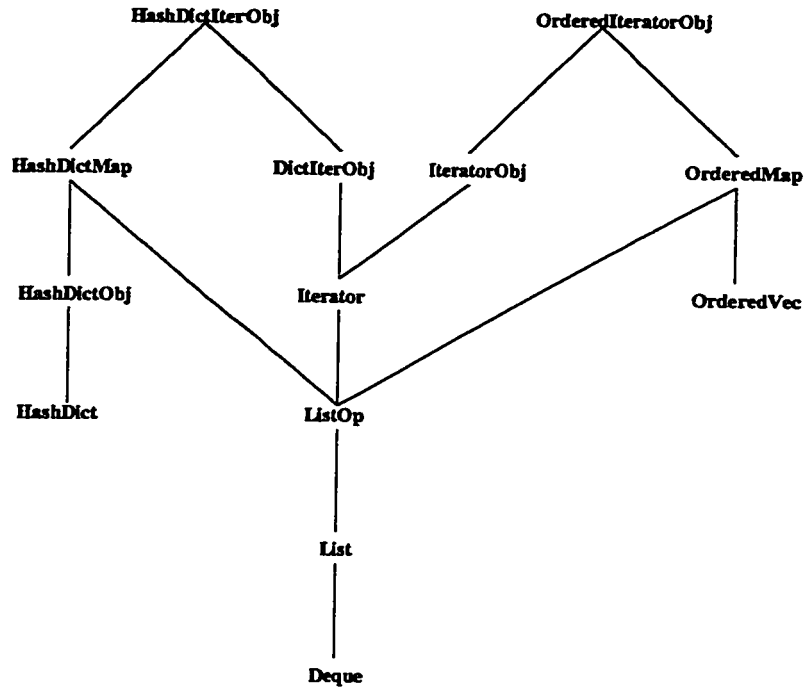


Figure 4: Reuse of LSL traits

3.5.2 Reuse of LSL Traits

Several traits from the LSL library [29] were reused in writing specifications of simple classes. However, the abstractions to be created for completely capturing the behavior of new classes required either minor modifications to library traits or new traits. The LSL traits **Iterator** and **IteratorObj** are generic iterators for respectively specifying a list of values and a list of objects. Hence, the iterator for each collection class need only specify a mapping function which would map the collection on to the list. Thus, the traits **HashMap**, **HashDictMap**, and **OrderedMap** were specified and used in conjunction with the **Iterator** for specifying **RWHashTableIterator**, **RWHashDictionaryIterator** and **RWOrderedIterator**. See Figure 4 for the hierarchy of trait inclusions in creating **HashDictIterObj** and **OrderedIterObj** traits used in the specifications of **RWHashDictionaryIterator** and **RWOrderedIterator** respectively.

Similarly, the abstraction for **RWHashDictionary** specified in the trait **HashDict** was reused in specifying the abstraction for the **RWBag** (please refer to trait **Bag_hashdict** in [2]). There are situations when either a direct reuse or an indirect

reuse through mapping functions are not possible. The case in point is the requirement for the class **RWHashDictionaryIterator** where an iterator for a list of tuples (key object, value object) iterating only on key objects has to be abstracted in the LSL tier. We cannot reuse **IteratorObj** here, and so we created a new LSL trait called **DictIterObj** to fully capture the semantics of **RWHashDictionaryIterator** class. For further details on the reuse of LSL traits we refer the reader to the reports [1, 2].

Declaration	Sort of x (x is global)
T x	Obj[T]
const T x	ConstObj[T]
T & x	Obj[T]
const T & x	ConstObj[T]
T & const x	Obj[T]
T * x	Obj[Ptr[Obj[T]]]
const T * x	Obj[Ptr[ConstObj[T]]]
T * const x	ConstObj[Ptr[Obj[T]]]
T x[3]	Arr[Obj[T]]
const T x[3]	Arr[ConstObj[T]]
IntList x	ConstObj[IntList]
int x(int i)	ConstObj[cpp_function]

Table 1: Sorts of Global Variables

Declaration	Sort of x (x is formal parameter)
T x	T
const T x	T
T & x	Obj[T]
const T & x	ConstObj[T]
T & const x	ConstObj[T]
T * x	Ptr[Obj[T]]
const T * x	Ptr[ConstObj[T]]
T * const x	Ptr[Obj[T]]
T x[]	Ptr[Obj[T]]
const T x[]	Ptr[ConstObj[T]]
IntList x	Val[IntList]

Table 2: Sorts of Formal Parameters

Chapter 4

Specifying Object Collaborations - A Formal Approach

The OMT notation [52] uses object diagrams to model the static structure of entities in the design, and describes the collaborations using event traces or Object Collaboration graphs (OCGs). Specification of object interfaces are given in pseudocode. These approaches are not formal, and do not provide a clear description of object dependencies and the inter-object behaviors that are maintained in the collaboration. Limitations of the *Contract* approach were discussed in Chapter 2.

A formal approach to specifying object collaborations must include the following:

- A specification of the structural and integrity constraints among the collaborating objects.
- A specification of the roles of the collaborating objects in a black-box fashion. This would also implicitly specify the interobject behavior of the collaboration.
- A specification of the interactive behavior in terms of the operation sequences and flow of control. This describes the manner in which objects collaborate to provide the interobject behavior.

It is clear that the specification of roles depends on the specification of structural aspects of the collaboration. Specifying the interaction between objects in the collaboration requires the services provided in the role specifications of objects in the collaboration. We have captured these details in a three-tiered, layered specification language.

4.1 Layer 1 – LSL Traits

The first layer specifies the structural aspects of the collaboration. Besides specifying the data models for the objects in the collaboration, it also specifies the *states of interest* of each object, and the cardinality constraints of the relationships among the collaborating objects. We use the Larch Shared Language LSL for specification of this layer.

4.2 Layer 2 – Role Specification

The second layer uses the constructs defined in Layer 1, and identifies the services required to specify the roles of the collaborating objects to specify their externally observable (interobject) behavior. The role specification for an object includes those services in the interface of the object which take part and are pertinent to the collaboration between the object and its collaborators. For each operation in the object's role specification, the specification consists of a *header* and a *body*. The header specifies the name of the operation, the names and types of parameters, as well as the return type, and uses exactly the same notation as in Larch/C++. The body of the specification consists of an **ensures** clause as well as optional **requires** and **modifies** clauses.

The **requires** and **ensures** clauses specify the pre- and post-condition of functions in the role specifications. The identifier *self* in the pre- and post-condition assertions denotes the object that receives the message corresponding to the specified method. The **modifies** clause lists those objects whose value may change as a result of executing the operation. An omitted **requires** clause is interpreted to mean that no object is modified by the corresponding method (neither *self* nor any parameter objects).

The link between an object's role specification and the trait specification is indicated by the clause **uses**. The *used trait* provides the names and meanings of the operators referred to in the pre- and post-conditions of an object's role specification. The **uses** clause also specifies the *role to sort* mapping and indicates the trait that specifies the abstract values of the objects involved in the role specification (for example, *self* and parameter objects).

4.2.1 V-action, S-action, O-action.

We can categorize the possible actions of a method in the role specifications of an object as follows:

- V-action: A method which returns a *value* belongs to this category.
- S-action: A method which *changes* the abstract *state* of an object's environment. By an object's environment, we shall mean any object other than the given object (which is related in some way to the given object).
- O-action: A method which *changes* the abstract *state* of that particular object belongs to this category.

Our V-action and O-action categories correspond to the V-function and O-function categories of [5]. However, the characterization differs from that of [5] in many important ways: (i) We use this characterization as a means to define what the behavior of a module is rather than to write a specification of the behavior of a particular module. (ii) We use abstract values to denote the values of method parameters and return values. This makes it possible to consider methods that takes as parameter or return values of complex objects rather than just basic values like integer and floating point numbers. (iii) In addition to considering methods which modify the internal state of an object, we also consider methods which modify the state of external environment. This makes it possible to provide a more precise definition of the role behavior.

We use the above category to classify the methods of a class. For example, a method which belongs to the O-action and V-action categories is referred to as an OV-method. We assume that a method which belongs to the S-category must always also belong to the O-category, so that any change in the environmental state of an object is reflected by a change in the abstract state of an object in the environment. Therefore, there can be no S-methods, only OS-methods or OSV-methods. The possible combinations for a method are O, O-V, O-S, V, and O-S-V. Intuitively, it is clear that these can be reduced to *canonical combinations* O, O-S, and V. This is because, it can be shown that any method which belongs to any other combination (ie. O-V, O-S-V) can be reduced to two methods each belonging to a canonical combination. That is, any method which belongs to the V category and to some other category can be split into two methods one of which is a pure function (V

portion) and another which is a procedure (the remaining O or O-S portion). Details of formal proofs of this would require defining the semantics of the object-oriented programming language being considered. To circumvent this and to avoid having to commit to a specific programming language, we choose to accept the validity of our assumption as an axiom.

Unlike Larch/C++ interface specifications it is not necessary for the documenter to specify all the services (including the constructor, destructor or copy constructor) but only those services that characterize the role of the object in the collaboration must be stated. For instance, role specifications of a document class collaborating with a file system would include only those services that participate and are relevant to the collaboration e.g. save and load a file. The document may also be interacting with a view object, but this has no relevance to the collaboration with the file system. In view of the discussions in the last section, we assert that it is possible for the role specifications to be presented only with pure O-actions, OS-actions and V-actions. We further make it mandatory that the role specifications specify all the pure O-actions, OS-actions and V-actions that are necessary for describing the object collaborations. Though this might introduce redundancy in role specifications when the same object plays several roles, we shall soon see how this helps in formalising object interactions.

4.3 Layer 3 – Collaboration Specification

The third layer uses the constructs specified in Layer 1 and Layer 2 to provide a specification of the interaction among the collaborating objects.

The interaction specifications for Layer 3 are provided by defining composition operators for pure O-action, V-action and OS-actions. Let O_1 and O_2 be two objects for which specification in Layer 1 and Layer 2 have been written. The collaboration of O_1 and O_2 can be described by a sequence of steps where in each step an interface action P of O_1 and an interface action Q of O_2 interact. The interface action interactions are the following:

- \wedge - Parallel Operation:

$R \triangle P \wedge Q$ defines an “independent collaboration”. The action of P on O_1 and the action of Q on O_2 can proceed independently. The semantics is:

$$pre(R) \Rightarrow pre(P) \wedge pre(Q)$$

$$post(P) \wedge post(Q) \Rightarrow post(R).$$

We extend the above to distributed parallel operation: $R \triangle \wedge P_i$ and conditional parallel operation: $R \triangle \wedge \{A(O_i) : P_i\}$. The semantics of $R \triangle \wedge P_i$ is as follows:

$$pre(R) \Rightarrow pre(P_1) \wedge pre(P_2) \wedge \dots$$

The semantics of $R \triangle \wedge \{A(O_i) : P_i\}$ is given by:

$$pre(R) \Rightarrow pre(P_1) \wedge pre(P_2) \wedge \dots$$

where each P_i is an interface action on an object O_i and O_i satisfies the predicate $A(O_i)$.

• ; - Sequential Composition:

$R \triangle P;Q$ defines a sequential collaboration. That is, the action Q of O_2 follows the completion of action P of O_1 . The semantics is

$$pre(R) \Rightarrow pre(P)$$

$$post(P) \Rightarrow pre(Q).$$

The following scenarios determine the post-condition of R . These are:

- case(i) $P = O, Q = OS$: $post(P) \wedge post(Q) \Rightarrow post(R)$
- case(ii) $P = OS, Q = O$: $post(P) \wedge post(Q) \Rightarrow post(R)$
- case(iii) $P = O$ or $OS, Q = V$: $post(P) \wedge post(Q) \Rightarrow post(R)$, and the value returned by V action is also the value returned by R .
- case(iv) $P = V, Q = O$ or OS : $post(Q) \Rightarrow post(R)$ and value returned by V -action is passed to the formal parameter of the method Q . In this case we denote R as $R \triangle P;Q@par$, where par represent the result of P .
- case(v) $P = O, Q = O$: $post(Q) \Rightarrow post(R)$. However, in this case the post-condition of Q must be evaluated by substituting the post-state of P for pre-state of Q (ie. $self^{\wedge}$ in post-condition of Q is actually $self^{\prime}$ of post-condition of P).

One can similarly specify the case for OS .

- [] - Choice Operator:

$R \triangle P [] Q$ defines the choice operator. That is, the choice of action P of O_1 , or action Q of O_2 depends upon which pre-condition evaluates to true. The semantics is:

$$(pre(R) \Rightarrow pre(P)) \Rightarrow (Post(P) \Rightarrow Post(R)) \\ \vee (pre(R) \Rightarrow pre(Q)) \Rightarrow (Post(Q) \Rightarrow Post(R)).$$

- $A? [] Not A?$ - if then else Operator:

$R \triangle A? P [] Not A? Q$ defines the if then else operator. A denotes an assertion defined in Layer 1 which is evaluated on the visible states of the object O_1 and O_2 . Informally the semantics is that, if the assertion A evaluates to true, then action P on O_1 is taken, else Q on O_2 is taken. Formally,

$$((pre(R) \wedge A \Rightarrow pre(P)) \Rightarrow (Post(P) \Rightarrow Post(R)))$$

or,

$$((pre(R) \wedge Not A \Rightarrow pre(Q)) \Rightarrow (Post(Q) \Rightarrow Post(R))).$$

In cases where else condition is not required, we specify $R \triangle A? P []$. In this case, if the assertion A evaluates to true, then action P on O_1 is taken, otherwise action P on O_1 is not taken.

Note that an operation of the form $R1 \triangle O; R$, where O stands for a pure O -action, would mean that the assertion A is evaluated in the post-state of O .

- *While A do* - Iterator

$R \triangle While A do P$ defines the repeat until operator. That is the action P on an object O_1 is repeated until the assertion A is false in the post-state of P . The semantics is:

$$pre(R) \wedge A \Rightarrow pre(P) \\ Post(P) \Rightarrow pre(R) \\ pre(R) \wedge Not A \Rightarrow TRUE$$

To summarise, the form of interaction specification is the name of object followed by a colon, and expressions of the form $R \triangle P Op Q$. The semantics of this expression is that the invocation of an interface method R in the *role specification* of

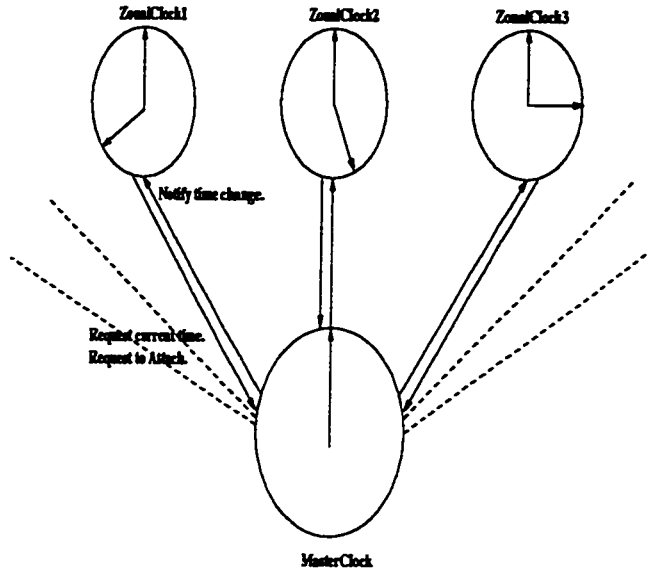


Figure 5. MasterClock – ZonalClock Micro-Architecture

the given object results in the invocation of interface action P , and interface action Q as per the semantics of Op operator. If method P is invoked on an object O different from the *self*, then P is denoted as $O.P$.

4.4 A Clock Example

In this section we provide a brief description of a micro-architecture that is a concrete instance of the *Observer* pattern [25] (p. 293).

4.4.1 Problem Description

Figure 5 illustrates the relationship between the master clock and zonal clock objects in this micro-architecture. The master clock object is responsible for maintaining the Greenwich Meridian Time, while the zonal clock objects display the time in their respective zones. The master clock object can exist independently of the zonal clocks, but each zonal clock depends on the master clock to maintain its zonal time. We thus have a one-to-many conditional relationship between master and zonal clocks. The master and zonal clocks together maintain an invariant that states that the zonal clock displaying the time in its zone is consistent with the master clock time.

In figure 6, we show an OMT [52] design diagram of the micro-architecture. This

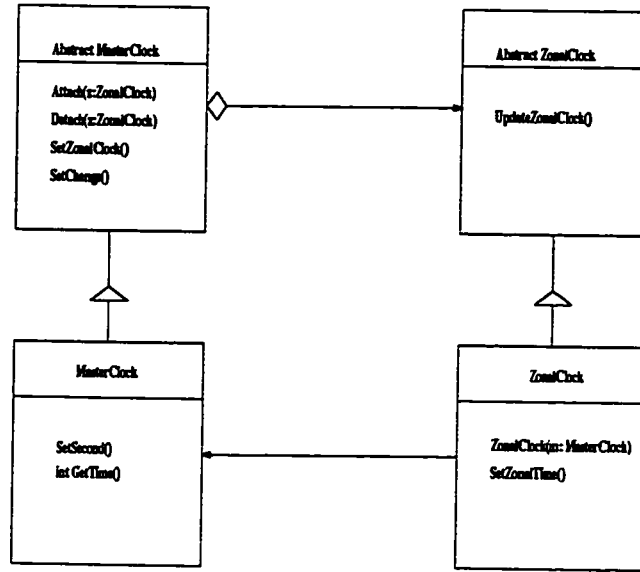


Figure 6: Design of the Micro-architecture

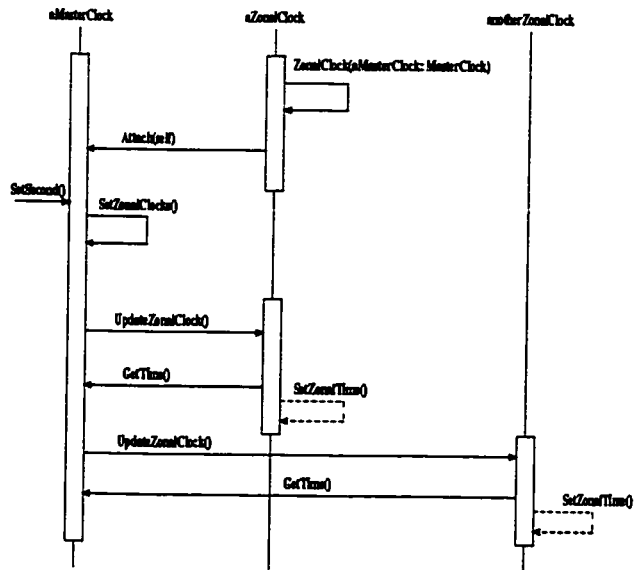


Figure 7: Object Collaboration Graph for the Clock Example.

design can be enhanced to include object references and Psuedocode. Figure 7 illustrates a collaboration scenario using an interaction diagram [6]. The interface of the participants in the micro-architecture relevant to this collaboration is described below:

MasterClock: This object provides an interface for attaching and detaching *ZonalClock* objects. The services include *SetSecond* method to update itself every second, *SetZonalClocks* to send notification to its *ZonalClocks* when its time changes, *SetChange* method to update its time and notify its *ZonalClocks* of time change, *GetTime* method to query the current time and maintain the Greenwich Meridian Time.

ZonalClock: This object provides an interface for creating a *ZonalClock* instance and attaching it to a *MasterClock* object, and provides an updating interface to keep its state consistent-with the *MasterClock's* state. *SetZonalTime()* method modifies the state of the *ZonalClock* to make it consistent with the *MasterClock's* time, and *UpdateZonalClock* method updates the *ZonalClock* object depending on whether an update is necessary.

The *MasterClock* notifies its *ZonalClocks* whenever a time change occurs, that is whenever its time is updated. After being informed of a change in the *MasterClock*, the *ZonalClock* object queries the *MasterClock* for the current time. If the *ZonalClock* object observes a time change then it updates itself to make its state (the zonal time) consistent with the *MasterClock's* time. Each *ZonalClock* object attaches itself to a *MasterClock* object upon its creation.

We have specified all the interface methods that are sufficient to describe the object collaborations in the micro-architecture.

4.4.2 Documenting a Solution

A micro-architecture specification involves the specification of the collaborations among the objects within the architecture. This must include three components:

- 1) *Specification of the one-to-many relationship*: There exists a one-to-many relationship between master clock object and the zonal clock objects. The integrity constraint on this relationship is that the zonal clock time must be consistent with the master clock time in its zone. Also, note that a zonal clock object cannot exist independently, and must be attached to master clock, while the

converse is not true. The integrity constraint characterizes the *states of interest* of the participating object in the collaboration. For instance, the name of a zonal clock (which is the name of the zone) does not affect the invariant between the zonal clock and the master clock to which it is attached.

- 2) *Communication protocol*: The services provided by the master clock object and the zonal clock objects relevant to their collaboration are part of the communication protocol. These services characterize the roles played by the master clock and zonal clock objects participating in the collaboration. These include the services that the master clock object invokes on itself to carry out a request, as well as services requested from other objects in the collaboration. For instance, the service that creates a zonal clock object forms part of role specification of the zonal clock object since it interacts with the master clock to attach itself to the master clock. Similarly, methods to modify the *states of interest* also form part of the roles.
- 3) *Service interactions*: The interaction between services provided in their roles, must be specified in terms of operation sequences, and flow of control. These specify the state transformations of the object collaboration.

4.5 Specification of the Case Study

We now illustrate the specification of the Clock example in our three-tiered specification language.

4.5.1 Layer 1 Specification

In this layer we specify the abstract states of *MasterClock* object and *ZonalClock* objects using LSL traits, the language of Layer 1. Since both *MasterClock* and *ZonalClock* maintain time, we first specify the *Time sort*. This is followed by a specification of the abstract values of the *ZonalClock* object, a specification of the relationship between the *MasterClock* and *ZonalClock* objects and the invariant properties of this relationship.

Time Specification.

The *Time* trait includes the traits *TotalOrder(Time)* and *Integer*. This is shown in the *includes* clause. The *TotalOrder* trait specifies formally the total ordering of the abstract values of *Time*. Similarly, the *tuple* specifies *Time* as a record structure. The signature of the *Time* trait introduces the following functions:

- *current_time*: Returns the current time.
- *convert*: Converts the given time into an integer.
- *reconvert*: Converts the given integer to time.
- *suc*: Given a time unit, returns the next time.
- *pred*: Given a time unit, returns the previous time.
- *inc*: Given a time and an integer, increments the time by the given number of seconds, and returns the new time.
- *dec*: Given a time and an integer, decrements the time by the given number of seconds, and returns the new time.
- *max*: Given two values of time, returns the maximum of the two.
- *min*: Given two values of time, returns the minimum of the two.
- \leq : Given two values of time, returns true if one precedes another and false otherwise.
- \geq : Given two values of time, returns true if one succeeds another and false otherwise.
- *isValid*: Given a value of time, returns true if the value is a valid time and false otherwise.

Time : trait

includes *TotalOrder(Time)*, *Integer*

Time tuple of hour, minute, second : *Int*

introduces

$current_time : \rightarrow Time$
 $convert : Time \rightarrow Int$
 $reconvert : Int \rightarrow Time$
 $suc : Time \rightarrow Time$
 $pred : Time \rightarrow Time$
 $inc : Time, Int \rightarrow Time$
 $dec : Time, Int \rightarrow Time$
 $max : Time, Time \rightarrow Time$
 $min : Time, Time \rightarrow Time$
 $- \leq - : Time, Time \rightarrow Bool$
 $- \geq - : Time, Time \rightarrow Bool$
 $isValid : Time \rightarrow Bool$

asserts

Time partitioned by convert
 $\forall t, t_1 : Time, h, m, s : Int, i : Int$
 $isValid(current_time)$
 $isValid(t) == convert(t) > 0$
 $isValid(t) == (t.hour \geq 0 \wedge t.hour < 24) \wedge$
 $(t.minute \geq 0 \wedge t.minute < 60) \wedge$
 $(t.second \geq 0 \wedge t.second < 60)$
 $convert(t) == (3600 * t.hour)$
 $+ (60 * t.minute) + t.second$
 $reconvert(convert(t)) == t$
 $suc(t) == reconvert((convert(t) + 1))$
 $pred(t) == reconvert((convert(t) - 1))$
 $inc(t, i) == reconvert((convert(t) + i))$
 $dec(t, i) == reconvert(convert(t) - i)$
 $t \geq t_1 == convert(t) \geq convert(t_1)$
 $t \leq t_1 == convert(t) \leq convert(t_1)$
 $max(t, t_1) = t == t \geq t_1$
 $min(t, t_1) = t == t \leq t_1$

implies

$\forall t : Time$

$$\text{succ}(\text{pred}(t)) == t$$

ZonalClock Specification.

This trait specifies the abstract values of the *ZonalClock* object. Since the *ZonalClock* object also maintains time, the abstract value of the *ZonalClock* object's time contains an abstract value *ZonalTime* specified by the *Time* sort. Further, the abstract value of the *ZonalClock* also includes its name and the standardOffset of the zone. A brief explanation of its signature follows:

- **Convert**: Given *Time* and *ZonalClock*, returns a *ZonalClock* value whose *ZonalTime* is equal to the given *Time* incremented by the standardOffset of the given *ZonalClock*.
- **isConsistent**: Given *Time* and *ZonalClock*, returns true if the *ZonalTime* and given *Time* are consistent and false otherwise.

ZonalClock : **trait**

includes *Integer*, *String*, *Time*

ZonalClock **tuple of** *standardName* : *String*,
standardOffset : *Int*, *ZonalTime* : *Time*

introduces

Convert : *Time*, *ZonalClock* → *ZonalClock*

isConsistent : *Time*, *ZonalClock* → *Bool*

asserts

$\forall z : \text{ZonalClock}, t : \text{Time}$

$\text{Convert}(t, z).\text{standardName} = z.\text{standardName}$

$\text{Convert}(t, z).\text{standardOffset} = z.\text{standardOffset}$

$\text{Convert}(t, z).\text{ZonalTime} = \text{reconvert}(\text{convert}(t) + z.\text{standardOffset})$

$\text{isConsistent}(t, z) == \text{if } (z.\text{ZonalTime} = \text{reconvert}(\text{convert}(t)$

$+z.\text{standardOffset})) \text{ then true}$

else false

implies

$\forall z : \text{ZonalClock}, t : \text{Time}$

$\text{isConsistent}(t, \text{Convert}(t, z))$

MasterZonal Specification.

This trait specifies the *MasterClock* object, *ZonalClock* objects and the structural and invariant properties of their relationship. A brief description of the signature follows:

- *Displays_ZonalTime_Of*: Given a *ZonalClock* object, returns the *MasterClock* object to which it is attached. The *ZonalClock* object depends on the *MasterClock* object for its current *ZonalTime*. The totality of this function specifies the constraint that a *ZonalClock* object cannot exist independently of the *MasterClock*.
- *Dept_ZonalClocks*: Given a *MasterClock* object, returns the possibly empty set of attached *ZonalClock* objects that depend on the *MasterClock* object for their *ZonalTime*.
- *ReflectsZonalTime*: Returns true if in the given state, an invariant holds among the given *MasterClock* and *ZonalClock* objects and false otherwise.

The *MutableObj* trait specifies the sort of mutable objects. *MutableObj(ZonalClock)* thus specifies the sort of mutable *ZonalClock* objects viz. $\text{Obj}[ZonalClock]$.

MutableObj(Time, MasterClock for Obj[Time]) specifies the sort $\text{Obj}[Time]$, which is the sort of objects whose abstract values are specified by the *Time* sort. Note that *MasterClock for Obj[Time]* renames $\text{Obj}[Time]$ to *MasterClock*. Thus the sort of *MasterClock* objects is *MasterClock*.

MasterZonal : **trait**

includes *MutableObj(ZonalClock)*,
MutableObj(Time, MasterClock for Obj[Time]),
Set(Obj[ZonalClock], Set[Obj[ZonalClock]])

introduces

Displays_ZonalTime_Of : $\text{Obj}[ZonalClock] \rightarrow \text{MasterClock}$
Dept_ZonalClocks : $\text{MasterClock} \rightarrow \text{Set}[\text{Obj}[ZonalClock]]$
ReflectsZonalTime : $\text{MasterClock}, \text{Obj}[ZonalClock], \text{State} \rightarrow \text{Bool}$

asserts

$\forall m : \text{MasterClock}, z : \text{Obj}[ZonalClock], st : \text{State}$
 $\text{Dept_ZonalClocks}(m) \geq 0$

$Displays_ZonalTime_Of(z) = m \implies z \in Dept_ZonalClocks(m)$

$ReflectsZonalTime(m, z, st) \implies (Displays_ZonalTime_Of(z) = m) \wedge$
 $isConsistent(m!st, z!st)$

4.5.2 Layer 2 Specification

We now specify the roles of the *MasterClock* object and *ZonalClock* object in the micro-architecture using the specification language of Layer 2 of our language.

MasterClock Role Specification.

MasterClock Role Spec:

uses: MasterZonal

Attach(z: ZonalClock)

```
{
    modifies Dept_ZonalClocks(self);
    ensures z ∈ Dept_ZonalClocks(self);
}
```

Detach(z: ZonalClock)

```
{
    requires z ∈ Dept_ZonalClocks(self);
    modifies Dept_ZonalClocks(self);
    ensures z ∉ Dept_ZonalClocks(self);
}
```

int GetTime()

```
{
    ensures result = convert(current_time(self\any));
}
```


SetSecond()

```
{  
    modifies self;  
    ensures  $self' = suc(self^)$ ;  
}
```

SetZonalClocks()

```
{  
    requires  $Dept\_ZonalClocks(self) \neq \{\}$ ;  
    modifies  $contained\_objects(Dept\_ZonalClocks(self), pre)$ ;  
    ensures  $\forall z : Obj[ZonalClock](z \in Dept\_ZonalClocks(self) \Rightarrow$   
         $ReflectsZonalTime(z, post))$ ;  
}
```

SetChange()

```
{  
    requires  $Dept\_ZonalClocks(self) \neq \{\}$ ;  
    modifies  $self \wedge contained\_objects(Dept\_ZonalClocks(self), pre)$ ;  
    ensures  $self' = suc(self^)$   $\wedge \forall z : Obj[ZonalClock]$   
         $(z \in Dept\_ZonalClocks(self) \Rightarrow ReflectsZonalTime(z, post))$ ;  
}
```

ZonalClock Role Specification.

ZonalClock Role Spec:

uses: MasterZonal

ZonalClock(m: MasterClock)

```
{  
    contracts self;  
    ensures  $Displays\_ZonalTime\_Of(self) = m$ ;  
}
```

```

UpdateZonalClock()
{
    modifies self;
    ensures ReflectsZonalTime(Display_ZonalTime_Of(self), self, post);
}

```

```

SetZonalTime(i: Int)
{
    modifies self;
    ensures self' = Convert(reconvert(i), self');
}

```

4.5.3 Layer 3 Specification

MasterClock:

$\text{SetZonalClocks()} \Delta \wedge \{\forall z : \text{Obj}[\text{ZonalClock}] z \in \text{Dept_ZonalClocks}(self) : z.\text{UpdateZonalClock}()\}$

$\text{SetChange()} \Delta \{\text{SetSecond}(); \text{SetZonalClocks}()\}$

ZonalClock:

$\text{ZonalClock}(m: \text{MasterClock}) \Delta \{m.\text{Attach}(self)\}$

$\text{UpdateZonalClock()} \Delta \{\text{Displays_ZonalTime_Of}(self).\text{GetTime}()\}$

$\text{ReflectsZonalTime}(self, \text{Displays_ZonalTime_Of}(self), st)?\text{SetZonalTime}(i: \text{Int})@\text{Int}[\dots]$

Chapter 5

Formalizing Design Pattern Documentations

The goal of this chapter is to study the applicability of formal approaches in documenting design patterns. We intend to study the extent to which formal approaches can improve the understandability, evaluation, and reuse of such documentation, by making design patterns easily comprehensible and provide a means to evaluate the design pattern. The basic concepts and the motivation for formalizing design pattern documentations was discussed in Chapter 1. In this chapter we first critically review certain existing documentation styles for describing design patterns. We then evaluate these approaches both for their style as well as for their content from the standpoint of formalizing them. The first step towards formalizing design pattern documentations, is to identify essential components of such documentation that should be formalized in order that design patterns may be reused.

5.1 Describing Design Patterns - An overview

There is yet no consensus on a documentation style for describing design patterns. We shall present in this section the documentation styles of Gamma et. al [25] (also called GOF style) and Schmidt's style [54].

5.1.1 Describing Patterns - GOF Style

Design patterns in the catalog of Gamma et al. [25] are described in a consistent format. Each pattern is divided into sections according to the following template.

- **Pattern Name and Classification:** The pattern's name conveys succinctly the essence of pattern. A good name is vital because it will become part of your design vocabulary. The design patterns are classified according to two criteria. The first criterion, called **purpose**, reflects what a pattern does. Patterns can have either **creational**, **structural**, or **behavioral** purpose. Creational patterns concern the process of object creation. Structural patterns deal with the compositions of objects or classes. Behavioral patterns characterize the way in which classes or objects interact and distribute responsibility.

The second criterion is called **scope**, which specifies whether the pattern applies primarily to classes or to objects. Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static – fixed at compile time. Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Creational class patterns defer some part of object creation to the subclasses, while creational object patterns defer it to another object. The structural class patterns use inheritance to compose classes, while the structural object patterns describe ways to assemble objects. The behavioral class patterns use inheritance to describe algorithms and flow of control, whereas behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

- **Intent:** A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem it addresses?
- **Also Known As:** Other names known for the pattern, if any.
- **Motivation:** A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help the user understand the more abstract description that follows.

- **Applicability:** What are the design situations in which the design pattern can be applied? What are the examples of poor designs that the pattern can address? How can you recognise these situations?
- **Structure:** A graphical presentation of the classes in the pattern using the notation based on Object Modelling Technique (OMT) [52]. Interaction diagrams [6] are used to illustrate sequences of requests and collaborations between objects.
- **Participants:** The classes and/or objects participating in the design pattern and their responsibilities.
- **Collaborations:** How the participants collaborate to carry out their responsibilities.
- **Consequences:** How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?
- **Implementation Code:** What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language specific issues?
- **Sample Code:** Code fragments that illustrate how the pattern might be implemented in C++ or Smalltalk.
- **Known Uses:** Examples of patterns found in real systems. At least two examples are provided from different domains.
- **Related Patterns:** What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

An Example

We present below the GOF style description of Abstract Factory pattern [25, 26]:

- **Pattern Name and Classification:** The name of the design pattern is Abstract Factory, it has object scope and creational purpose.

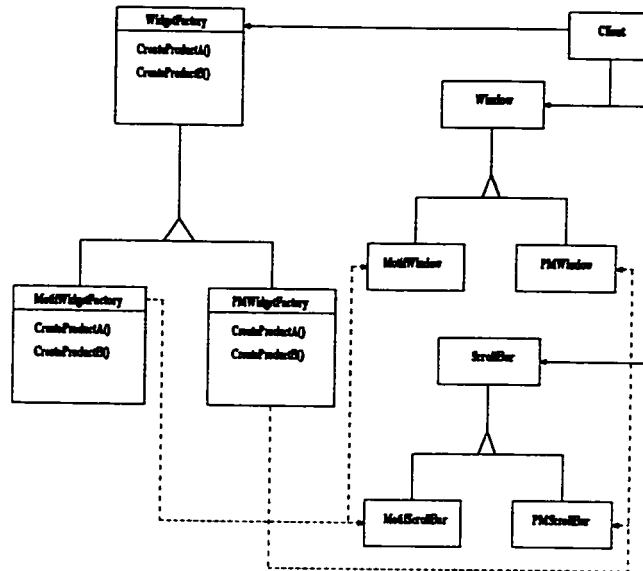


Figure 8: AbstractFactory - Contextual Example

- **Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Also Known As:** Kit
- **Motivation:** User interface toolkits supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feel standards support different appearances and behaviors for user interface “widgets” such as scroll bars, windows and buttons. To be portable across multiple look-and-feel standards, an application should not hard-code its widgets for a particular look-and-feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it harder to change the look and feel later.

This problem can be solved by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There’s also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. WidgetFactory’s interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients are not aware of the concrete classes they are using. Thus clients stay independent of the prevailing look and feel. There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate

widget for the look and feel. For example, the `CreateScrollBar` operation on the `MotifWidgetFactory` instantiates and returns a Motif scroll bar, while the corresponding operation on the `PMWidgetFactory` returns a scroll bar for the Presentation Manager. Clients create widgets solely through the `WidgetFactory` interface, and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, the clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

A widget factory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a `MotifWidgetFactory`.

- **Applicability:** Use the Abstract Factory when
 - a system should be independent of how its products are created, composed and represented.
 - a system should be configured with one of the multiple families of products.
 - a family of related product objects is designed to be used together, and the user should enforce this constraint.
 - the user may want to provide a class library of products, and want to reveal just their interfaces and not their implementations.
- **Structure:** Please refer to Figure 9.
- **Participants:**
 - **AbstractFactory** (Widget Factory)
 - * declares an interface for operations that create abstract product objects.
 - **ConcreteFactory** (`MotifWidgetFactory`, `PMWidgetFactory`)
 - * implements the operations to create concrete product objects.
 - **AbstractProduct** (`Window`, `ScrollBar`)
 - * declares an interface for a type of product object.
 - **ConcreteProduct** (`MotifWindow`, `MotifScrollBar`)

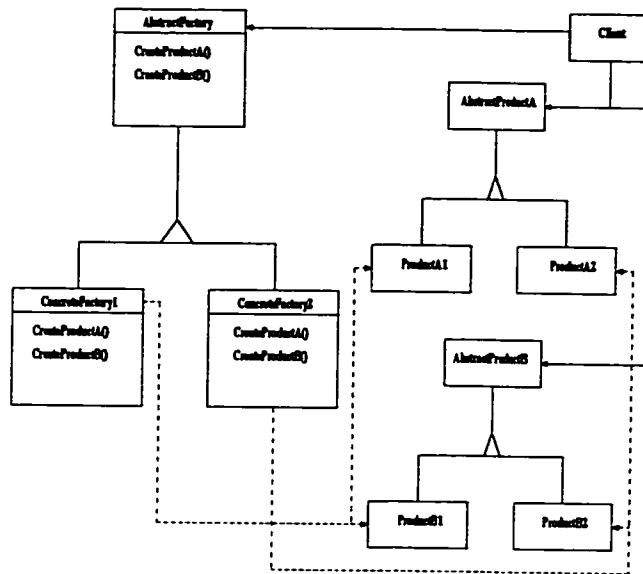


Figure 9: OMT Diagram For Abstract Factory Structure

- * defines a product object to be created by the corresponding concrete factory.
 - * implements the AbstractProduct interface.
- Client
- * uses only interfaces declared by AbstractFactory and AbstractProduct classes.
- **Collaborations:**
 - Normally a single instance of a ConcreteFactory class is created at runtime. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
 - AbstractFactory defers creation of product objects to its ConcreteFactory subclass.
 - **Consequences:** The Abstract Factory Pattern has the following benefits and liabilities:
 1. *It isolates concrete classes.* The Abstract Factory pattern helps to control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it

isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

2. *It makes exchanging product families easy.* The class of a concrete factory appears only once in an application – that is, where it's instantiated. This makes it easy to change the concrete factory used by an application. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once. In our user interface example, we can switch from Motif widgets to Presentation Manager widgets simply by switching the corresponding factory objects and recreating the interface.
 3. *It promotes consistency among products.* When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.
 4. *Supporting new kinds of products is difficult.* Extending abstract factories to produce new kinds of Products isn't easy. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all its subclasses. We discuss one solution to this problem in the implementation section.
- **Implementation:** A novel implementation is possible in Smalltalk. Because classes are first class objects, it is not necessary to have distinct ConcreteFactory subclasses to create variations in the products. Instead, it is possible to store classes that create these products in variables inside a concrete factory. These classes create new instances instead of the concrete factory. This technique presents variations in product objects at finer levels of granularity than by using distinct concrete factories. Only the classes kept in the variables need be changed.

We refer the reader to several other implementation issues discussed in detail in [25].

- **Sample Code:** An example to build a maze for a computer game is presented. A maze is defined as a set of rooms. A room knows its neighbors; possible neighbors are another room, a wall, or a door to another room.

The classes *Room*, *Door* and *Wall* define the components of the maze. Abstract Factory pattern can be applied for creating such mazes. The class *MazeFactory* can create components of the mazes. It builds rooms, walls, and doors between the rooms. It might be used by a program that reads plans for mazes from a file and builds the corresponding maze. Or it might be used by a program that builds mazes randomly. Programs that build mazes take a *MazeFactory* as an argument so that the programmer can specify the classes of rooms, walls, and doors to construct.

```
class MazeFactory {
public:
    MazeFactory();
    virtual Maze* MakeMaze() const
    {return new maze;}
    virtual Wall* MakeWall() const
    {return new Wall;}
    virtual Room* MakeRoom(int n) const
    {return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    {return new Door(r1, r2); } }
```

The class *MazeGame* creates the maze. The member function *CreateMaze* in this class builds a small maze consisting of two rooms with a door between them.

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
Maze* aMaze = factory.MakeMaze();
Room* r1 = factory.MakeRoom(1);
Room* r2 = factory.MakeRoom(2);
Door* aDoor = factory.MakeDoor(r1, r2);
aMaze→AddRoom(r1);
aMaze→AddRoom(r2);
r1→SetSide(North, factory.MakeWall());
```

```

    r1→SetSide(East, aDoor);
    r1→SetSide(South, factory.MakeWall());
    r1→SetSide(West, factory.MakeWall());
    r2→SetSide(North, factory.MakeWall());
    r2→SetSide(East, factory.MakeWall());
    r2→SetSide(South, factory.MakeWall());
    r2→SetSide(West, aDoor);
    return aMaze;
}

```

Similarly, one can create *EnchantedMazeFactory*, a factory for enchanted mazes, by subclassing *MazeFactory*. *EnchantedMazeFactory* will override different member functions and return different subclasses of Room, Wall, etc. Due to space considerations, we refer the reader to [25] for more details on this example.

- **Known Uses:** InterViews uses the “Kit” suffix [43] to denote abstract factory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending upon the layout desired.

ET++ [60] employs the Abstract Factory pattern to achieve portability across different window systems (X Windows and SunView, for example). The WindowSystem abstract base class defines the interface for creating objects representing window system resources (for example MakeWindow, MakeFont, MakeColor). Concrete subclasses implement the interfaces for a specific window system. At run-time ET++ creates an instance of a concrete WindowSystem subclass that creates system resource objects.

- **Related Patterns:**

Abstract Factories are often implemented using factory methods (please refer factory Method pattern in [25]), but they can also be implemented using Prototype pattern [25].

A concrete factory is often a singleton (please refer Singleton pattern in [25]).

5.1.2 Describing Patterns - Doug Schmidt's Style

Schmidt's style [54] emphasizes on articulation of non-functional forces that a design pattern resolves, besides recording the static and dynamic among the participants in the software design. We first present the style followed by an example in this style [54].

The main parts of this documentation style are the following:

- **Name:** Represents the name of the pattern.
- **Problem and context:** A brief description of the problem is provided. This is followed by a description of the context in which the pattern is used.
- **Force(s) addressed:** Describes a set of design issues that force the solution (ie. motivate the solution) proposed by the design pattern. -
- **Solution:** An abstract description of structure and collaborations in solution. These are presented typically using class diagrams, object diagrams and object interaction graphs in Booch style [6] and in informal textual notation. The solution solves the described problem in the given context by resolving the forces stated.
- **Consequences:** The positive and negative consequences of using the pattern are provided.

This description may be supplemented with sample code and examples of the design pattern usage in various systems.

An Example

We present below the Abstract Factory pattern [25] in the documentation style presented above.

- **Name:** Abstract Factory
- **Problem and Context:**
 - **Problem:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- Context: Same as in Motivation section in 5.1.1.
- **Forces:**
 - Clients use objects from only one family at any time.
 - Client of the families must not depend on an object's representation.
 - There may be more than one family of objects being used by a client.
- **Solution:** This is similar to the **Structure, Participants and Collaborations** sections in 5.1.1, with the exception that instead of an OMT diagram a Booch style [6] class diagram is presented.
- **Consequences:** These are also similar to the **Consequences** section in 5.1.1. 5.1.1

Besides the above two documentation styles, several other styles have also been proposed and we refer the reader to series of conference proceedings devoted to *Pattern languages of program design* [56]. Also, see the *Patterns Home Page* at [url:http://st-www.cs.uiuc.edu/users/patterns/patterns.html](http://st-www.cs.uiuc.edu/users/patterns/patterns.html). Among the documentation styles, the GOF style is the most popular and we shall be discussing this in detail in later sections.

5.1.3 Comparison of the Two Styles

In this section, our goal is to compare and contrast the two styles of documentation presented above, and identify the key components of a documentation style that are essential to reuse. These key components will then serve as a basis for a formal approach. We observe that the notable difference between the two styles is the articulation of the so called *forces*. The *problem* and *context* aspects of the two styles are identical. Schmidt's style emphasizes on *forces* that motivate the design *solution* provided by the design pattern. These forces characterize the additional requirements that the design pattern's solution must satisfy besides those stated in its problem. It is also clear that these forces arise in the context described in the **Problem and context** section in 5.1.2.

GOF style, on the other hand outlines these forces within the context of motivating example, and also as part of applicability conditions (see **Motivation and Applicability** sections in 5.1.1). For instance, in the Abstract Factory design pattern, the

Motivation section brings out a force that each WidgetFactory must enforce dependencies between the concrete widget classes (please refer section 5.1.1). However, this force is stated within the context of a specific example, which in this case is the creation of user interface widgets. The same force in Doug's style (please refer 5.1.2) is presented distinctly under **Forces** section and is stated abstractly (ie. in a general context). For the following reasons we believe the latter approach is more useful to a reuser:

- Since, the problem and solution aspects of a design pattern are highly abstract it is important to present the design issues also in abstract fashion. Obscuring them within a specific example leaves the reuser with the additional task of reading lengthy examples to identifying them.
- The forces represent some aspects of the design rationale underlying a design pattern.
- Stating the forces abstractly and distinct from the context does not confuse the reuser with irrelevant details he might otherwise encounter in the description of the specific example in the context.

Schmidt's style does not present applicability conditions. An analysis of the applicability conditions described in the **Applicability** section of patterns in [25] reveals that such conditions typically deal with :

- 1) situations that can cause changes to existing requirements or introduce new requirements.
- 2) situations other than those stated in the *problem* statement in which the pattern may still be used.
- 3) reiteration of the design issues considered in the design of the design pattern's solution (these are stated in the context of a specific example within the **Motivation** section of 5.1.1).

To a reuser 1) and 3) are more important since they describe aspects of design rationale. Situations in 2) are derived from the pattern writer's experience or the experience of other pattern users. To a user interested in understanding a design pattern in order to reuse it, knowing the rationale of pattern is more important than

seeing the situations in which it can be applied; in particular for novice users these situations may be difficult to understand and may appear speculative. We therefore believe that such issues can be presented as useful comments rather than form core of a pattern documentation. We shall therefore not deal with such situations in our approach.

The **Intent** section in 5.1.1 and **Problem** section in 5.1.2 describes the intent of the pattern. According to Gamma et al [25] the **Intent** must describe briefly the problem and how the problem is solved. We believe that it is easier to understand a problem when the solution details are not provided.

The **Consequences** aspects of both styles state the benefits and liabilities of using the design pattern. They state how the design pattern solution satisfies the design issues mentioned in the **Problem**, **Motivation**, and **Applicability** sections. Similarly, those in Schmidt's style mention how the **Forces** stated are resolved. In both cases, the resolution of design issues already stated are reiterated as positive consequences of the design. Sometimes, newer requirements are mentioned in the consequences. We believe that all the advantages of using a pattern can be stated as motivating forces for the design. However, it is important to mention how certain forces are resolved as part of consequences. For instance, in the Abstract Factory pattern, the **Consequence** section reiterates that it must be possible to exchange the product families. This is redundant since, the **Forces** section mentions that solution must make it easy to change between the families of product objects. Such redundancies must be avoided since a reuser who understands the forces motivating a design can be rest assured that these forces are indeed met by the solution. What is important however is to understand how the forces are met. This is at times clear from the description of the solution, while at times additional information on how the design pattern can be customized to meet new requirements needs to be provided in the **Consequence** section. Thus the consequences should only deal with customization of design pattern solution to meet new requirements or changes to existing ones and drawbacks of using the pattern.

It is clear from our discussion that documenting the problem solved by a pattern is at least as important, if not more as describing the pattern's solution itself. Though it is easier to see a pattern as a solution, as a technique that can be reused and adapted, it's harder to see when it is *appropriate* – to characterize the problem it solves and the context in which it is the best solution [25]. Characterizing the design problem

involves clearly outlining the design rationale underlying the pattern. Understanding the design rationale is crucial to understand and choose a right pattern to apply. Various documentation styles including those discussed above attempt to describe the rationale in various ways. From our discussion above, it follows that in GOF style this is spread over **Motivation**, **Applicability** and **Consequences** sections, and Schmidt's style, this is present in **Context**, **Forces** and **Consequences**. It is also clear from patterns in [25] that the design rationale underlying a pattern can be further classified as follows:

- design choices and assumptions made about the problem's context: we will call them *constraints* since they constrain the solution space for the **problem** addressed by the pattern. For example, the constraints in the Singleton pattern [25], can be stated as:

A global variable cannot be used to represent a singleton object since although it makes objects accessible, it does not prevent clients from instantiating multiple objects.

- requirements that are anticipated to arise in future. Often, the constraints considered in a design pattern imply certain anticipatory requirements. Still, it is better to state these requirements explicitly as *Reuse Requirements*. For example, the reuse requirements in Observer [25] can be stated as:

It should be possible to vary the type of observer objects without varying the subject in the design.

5.2 Template - A New Documentation Style

Based on these discussions we propose a template style for documenting design patterns. This template is motivated by our discussions, and the need to spell out in a non-redundant, and abstract fashion the aspects of the design pattern documentations that are crucial to their reuse. Our template will have a **Problem** section that describes briefly the problem statement without providing any details of the design pattern solution. The properties of the desired solution may be stated as **Constraints** which also describe design issues considered. **Reuse Requirements** explicitly state that the design should be customizable to meet new needs. Besides, the design rationale as described above, the pattern should explain how the **Reuse Requirements**

will be satisfied by the design pattern's solution. These are described under **Reuse Requirement Resolution** section following the **Solution**. As already discussed, we do not see the need for reiterating the advantages of the design pattern solution in terms of design constraints and reuse requirements it satisfies since these are already stated before the **Solution** and the reuser can expect these to be satisfied. We replace the positive consequences with the **Reuse Requirement Resolution** section. Then a section on **Drawbacks** is added. Each section in the proposed template is annotated with an example thereby making it easier to understand the pattern. The template we propose has the following format:

- **Name:** Represents the name of the pattern.
- **Problem:** A brief description of the problem.
Example Problem:
- **Constraints:** Description of design constraints (as explained above).
Example Constraints
- **Reuse Requirements:** Description of the reuse requirements as explained above.
Example Reuse Requirements
- **Solution:** A description of the solution in GOF style.
Example Solution
- **Reuse Requirement Resolution:** Description of the reuse requirements and how they are resolved by the solution.
- **Drawbacks:** The negative consequences of using the design pattern.
- **Implementation issues and Sample Code:** A description similar to the GOF or Schmidt style.

We consider our template approach to remain flexible so as to represent evolving concepts in design patterns. We believe this approach is also a better way of organizing the information towards a clear understanding of the design pattern notions.

We now present a few examples to illustrate this approach. The reader is requested to refer to [25] to compare our style with GOF and also to understand the relevant

sections. In these examples we shall omit details on implementation issues and sample code.

5.2.1 Abstract Factory

- **Name:** Abstract Factory.
- **Problem:** Clients want to create families of related or dependent *product* objects. Further, clients use product objects from only one family at any time.
Example: An application wants to create user interface widgets such as windows, scrollbars, and buttons belonging to various look-and-feel standards such as Motif and Presentation Manager.
- **Constraints:**
 - The clients of the families must not depend on the product object's representation. The clients should not create product objects from the families by specifying their class explicitly. For, doing so would make it harder for the client to change the product family later.
Example: To be portable across look-and-feel standards, an application should not hard code its widgets for a particular look-and-feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it harder to change the look and feel better.
- **Reuse Requirements:**
 - It should be possible to vary the family being used by a client.
Example: It should be possible for the application to vary the look-and-feel ie. switch between look-and-feel standards.
 - It should be possible to add new families of products.
It should be possible to port the application to new look-and-feel standards.
- **Solution:** This is identical to the **Structure, Participants and Collaborations** sections in 5.1.1.
Example: This is identical to the OMT diagram and illustration of the example in the Motivation section of 5.1.1

- **Reuse Requirements Resolution:**

- The class of the concrete factory appears only once in the application viz. where it is instantiated. The clients can be configured either statically or dynamically with the desired ConcreteFactory. In either cases, the clients require little or no modification. Changing the concrete factory automatically changes the product objects that will be used by clients. It is thus easy to vary the family of product objects being used by the clients.
- To add a new family of product objects, one has to define new factory class (representing the new family) and define product classes in the new family (representing the product objects of the family) and make them as sub-classes of AbstractFactory and Product classes respectively. Of course, this is possible only if the product objects in the new family are “related” to those in the existing family(or families).

- **Drawbacks:** Only the negative consequences of the pattern as described in 5.1.1 are provided.

5.2.2 Composite

- **Name:** Composite

- **Problem:** Compose objects into tree structures to represent part-whole hierarchies.

Example: Graphic applications such as drawing editors build complex diagrams by composing simple components.

- **Constraints:**

- Clients must be programmed to an uniform interface of primitive and composite structures and not to their individual implementations. This constraint is based on the assumption that clients treat primitive and composite structures uniformly *most of the time*. So, if the application is configured with their implementations, the application becomes unnecessarily complex. This constraint makes the client code simple.

Example: If the drawing editor application treats primitive and container

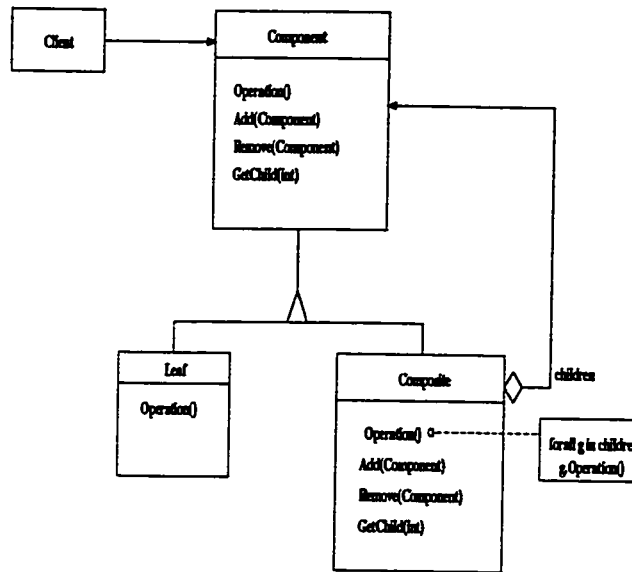


Figure 10: OMT Diagram For Composite Structure

objects differently, even if most of the time they are used identically, the application becomes unnecessarily complex.

- **Reuse Requirements:**

- It should be easy to define new types of primitive and composite objects without altering the code of existing classes or clients.

Example: It should be possible to add to new types of simple and complex drawing components without altering the code of the rest of the application.

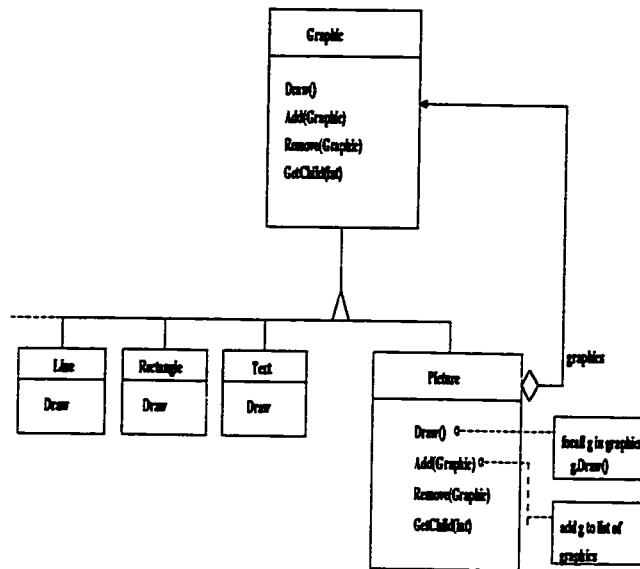
- It should be easy to vary the structure and composition of a primitive or composite object in the design.

Example: It should be easy to vary the implementation of a complex/simple component and vary the components within a complex component.

- **Solution:** This can be described similar to the **Structure, Participants and Collaborations** sections in 5.1.1. Please see Figure 10 [25].

Example: This can be described similar to the OMT diagram and illustration of the example in the Motivation section of 5.1.1. Please refer to Figure 11 [25].

- **Reuse Requirements Resolution:**



• Figure 11: Composite - Contextual Example

- To add a new types of primitive and composite objects, one has to define new leaf class (representing the new family) and define composite classes in the new family (representing the product objects of the family) and make these classes sub-classes of Component and Composite classes respectively. Of course, this may require that the interface of Composite is class be modified and the objects in the newly defined classes satisfy the constraints already stated.
- Since the Composite class defines all child-related operations in the Component interface, it is easy to vary its structure at run-time. Also, since clients only know the interface of Component they are unaffected by changes in implementation of Primitive and Composite.
- **Drawbacks:** The design pattern can make the design general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes the user may want a composite to have only certain components. With Composite, one has to perform run-time checks to enforce these constraints [25].

5.2.3 Observer

- **Name:** Observer

- **Problem:** Define a one-to-many dependency between objects such that when one object changes its state all its dependents are notified and their states are updated automatically.

Example: Consider a spreadsheet and barchart object depicting information in the same application data object using different presentations. When a user changes information in the spreadsheet, the barchart reflects the change immediately, and vice versa. The spreadsheet and barchart object are dependent on the data object and therefore should be notified of any change in its state.

- **Constraints:**

- The object and its dependents should not be tightly coupled since, doing so will not enable them to be reused independently. Loose coupling increases the chances of the two being reused independently.

Example: The application data object must not be tightly coupled with bar-chart and spreadsheet objects.

- **Reuse Requirements:**

- It should be possible to vary the number and type of dependent objects (type refers to the way they react to an update event).

Example: It should be easy to vary the number and type of user interfaces to the same data.

- **Solution:** This can be described similar to the **Structure, Participants and Collaborations** sections in 5.1.1. Please refer to Figure 12 [25].

Example: This is identical to the OMT diagram and illustration of the example in the Motivation section of 5.1.1. Please refer to Figure 13 [25].

- **Reuse Requirement Resolution:**

- To add new types of observers objects, define the new observer class with the Update operation and make it a subclass of the Abstract Observer Class. The abstract coupling between subject and observer ensures that no change in the Subject class is required. Thus, it is easy to define new types of Observer objects that can subscribe to the notification from the Subject.

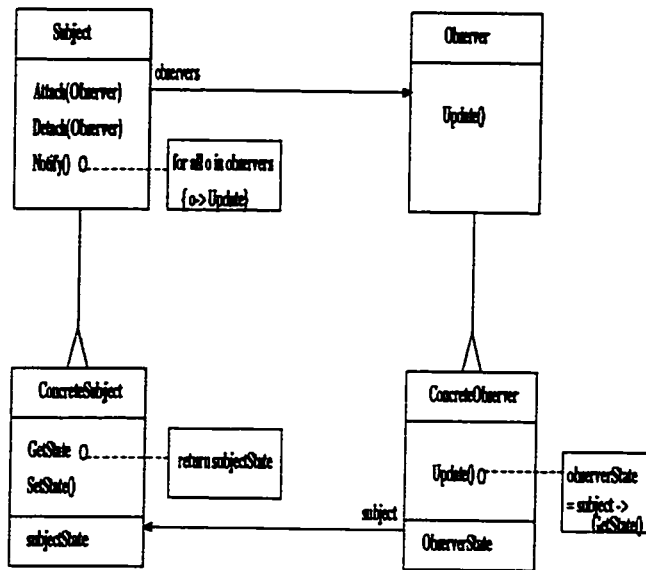


Figure 12: OMT Diagram For Observer Structure

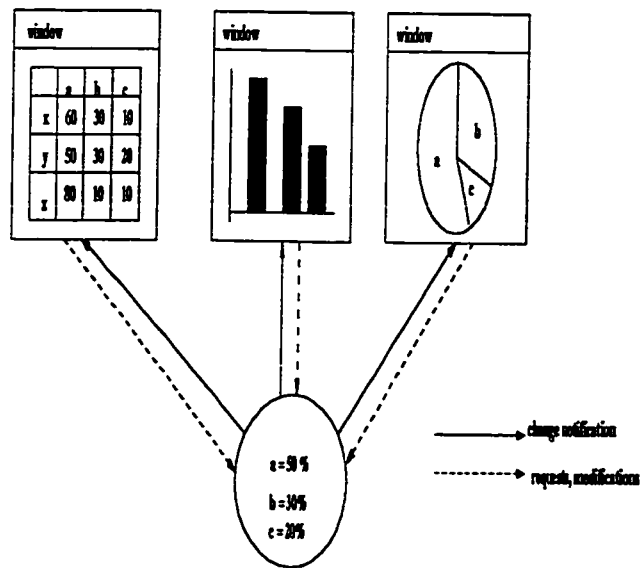


Figure 13: Observer - Contextual Example

- The interface of Subject has the Notify operation that broadcasts the notification to all the dependent observers (ie. observers that have subscribed to the notification). This gives one the freedom to add and remove observers at any time.
- **Drawbacks: *Unexpected Updates*.** Because observers have no knowledge of each other's presence, they can be blind to the cost of ultimately changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects. Moreover, dependency criteria that are not well-defined or maintained usually lead to spurious updates, which can be hard to crack down. This problem is aggravated by the fact that the simple update protocol provides no details on the causes that changed in the subject. Without additional protocol to help observers discover the causes of change, they may be forced to work hard to deduce the reasons for changes [25].

5.3 Towards A Formal Approach

One of the biggest challenges facing widespread reuse of design patterns is the ability to document them unambiguously. Design patterns, by virtue of their level of use, must be kept at a high level of abstraction. From the experience gained in reading the design patterns in [25] and producing the template style, we are lead to the next natural step wherein we address the two key issues: (1) what aspects of design patterns should not be formalized? and (2) what aspects of design patterns must be formalized?

The template documentation presented in the last section makes a clear distinction between the abstract aspects of the design pattern and its instantiation in a specific context. This distinction is crucial for reuse. However, the level of informality in template descriptions may give rise to ambiguities and inconsistencies. We discuss below these issues for each section of the informal template in 5.2.

5.3.1 Problem Section

The template provides a high-level description of the **Problem** and **Constraint** sections. This is done with a view of ensuring wide applicability of the pattern.

For instance, the **Problem** of Abstract Factory states that clients want to create families of related or dependent objects. This description does not specify the nature of relationship or dependency among the objects. Similarly, the notion of “family” of objects is also not clear. These notions can be understood only from an examination of abstractions present in the context (the motivating example in **Motivation** section of 5.1.1). However, the context captures these abstractions in the context of a specific example. So, there is a danger that a reuser may be caught in irrelevant details specific to the contextual example, and lose track of abstractions that are relevant to the design issue described in **Problem**.

For instance, it is clear from the contextual example of the Abstract Factory pattern that the abstractions MotifWindow, MotifScrollbar, PMWindow, PMScrollbar are relevant to the design issue described in the **Problem**. However the abstraction for MotifFactory is irrelevant since this pertains to the instantiation of the design pattern solution. If the abstractions that characterize the design problem addressed by a pattern are not discernible from the context, then the user has to rely on the abstract solution provided to understand the design problem. This is worse, since the problem and solution are at different levels of detail and abstraction. For instance, in the Abstract Factory pattern, a user can infer from the **Solution** that the product objects are partitioned in two types viz. AbstractProductA, and AbstractProductB. Further, they are related in the sense, that product objects in classes ProductA1 and ProductB1 can be created only by an instance of ConcreteFactory1. Similarly, the product objects in the classes ProductA2, ProductB2 can be created only by an instance of ConcreteFactory2. Further, the **Solution** makes it clear that clients can use objects from only one factory at any time. It is clear from the solution, that each ConcreteFactory class represents a family of objects. Suppose, that the **Problem** section describes a design issue which does not specify that product objects can be partitioned into such types and that the clients can use only one family of products at any time, the solution proposed is not a valid solution. For, if problem domain does not state that Products ProductA1, and ProductA2 have the same type, and yet the solution provides an inheritance hierarchy where ProductA1 and ProductA2 inherit from the abstract class ProductA, then this would be a violation of OO design principles.

Similarly, the **Problem** section in Composite pattern [25] states that the pattern composes objects into tree structures to represent part-whole hierarchies. Again, from

the abstract problem statement the meaning of “composes” is not clear. Does it mean that the pattern must define a representation of a complex object, or does it mean that it should provide a procedure for composing objects. It is also not clear how an object and its composition are related. These became clear only from the context. The context section outlines that users for most part request the same set of services from an object and its composition. It is also clear that the goal of the pattern is to provides a representation for the complex object in terms of its components.

The **Problem** of Observer pattern states that there is a one-to-many dependency between certain objects. The nature of dependency is not specified. However, the contextual example and collaboration sections under the **Solution** clearly spell out that there is a state dependency between such objects. Similar criticism applies to all patterns found in [25].

In summary, the informal description in the **Problem** section in any design pattern [25] does not identify the abstractions present in the problem. Such discrepancies can be identified by the reuser only through an analysis of the context and the solution. Sometimes, the motivating example is itself difficult to understand requires considerable amount of time to understanding. Identifying abstractions pertaining to the **Problem** from the solution might bias the reader with details pertaining to the solution. Besides, the **Problem** and **Solution** are at different levels of abstractions and detail. A formal approach can identify essential abstractions relevant to the design issue at hand without confusing the reader with irrelevant details in the context and also not biasing the reader with solution level details. We shall call such abstractions as “problem domain entities”.

5.3.2 Constraints Section

We have already stated that the **Problem** section of the formal documentation must specify the problem domain entities, their relationships and the task that should be achieved by the pattern in terms of these entities. The **Constraints** in our informal template describe the design choices and their rationale. The design choices constrain possible solutions of the design issue outlined in the **Problem**. Thus they specify the manner in which problem domain entities must be represented in the solution. For instance, the Abstract Factory pattern’s constraint makes a design choice that the clients must not be configured with the concrete classes of product objects they

may create at run-time. The problem domain entities identified in the **Problem** would include the concrete classes of such product objects. Then the design choice states that though clients may use objects from these classes, the clients do not create these objects statically by explicitly specifying that they are instances of the concrete classes. Similarly, the Composite pattern's constraint states that clients which use primitive and composite objects do so by assuming that they have the same interface. Since the primitive and composite classes have already been identified in the problem domain, this constraint can be stated in terms of these entities. Stating such design choices informally introduces several ambiguities. For instance, the design choice for Composite pattern states that clients assume the same interface. Again, the meaning of "assume" is not clear. Similarly, the Observer design pattern states that Subject and Observer classes must not be tightly coupled. Though, the meaning of coupling is still not clear in OO community (Gamma et al. define it as the degree to which objects "know" about each other), formalism can provide a more precise interpretation of this design choice: formally, it can be stated that the subject does not have a reference to an object of observer class.

It is thus clear that a formal approach to stating the design choices in the constraints is desirable. Further, since the design choices may be stated in terms of problem domain entities the constraints formalism can incorporate the formalism for the **Problem**. It is also clear that abstractions in the **Problem** and the design choices in the **Constraint** pertain to the problem domain of the design pattern. Though the design choices may be stated formally, the rationale for such choices cannot be stated formally as these are often descriptive and subjective. Thus formalizing constraints involves formalizing design choices made, in terms of the problem domain entities. The rationale for making such design choices cannot be formalized.

5.3.3 Reuse Requirement Section

Now, consider the **Reuse Requirements** stated in our template 5.2. These characterize the anticipatory requirements and aspects that should or should not vary in the proposed design. Consider for instance, such reuse requirements for the Composite pattern. One of them states that, it should be easy to vary the structure and composition of the composite object. Another states that it should be easy to add new primitive and composite objects to the proposed without altering the clients or other

objects in the proposed design structure. These statements cannot be formalized using the abstractions introduced in the **Problem** and **Constraint** sections since the informal semantics of *varying* the structure of a composite object or *adding* new types of primitive objects to the structure describe properties that are too vague and ambiguous to be formally stated. For instance, for a reader interested in understanding the design issue, the term *varying* can mean several things. It could mean that the solution allows us to vary composition of composite object at run-time or the implementation of the composite object at run-time. The semantics of *vary* is clear only from the specification of the **Solution**. It is clear that since the Composite class implements child-related operations in the Component interface, the composition of composite objects can be easily changed at run-time. Similarly, in the Observer pattern, the reuse requirement states that it should be easy to vary the type of observers. At the level of problem domain abstractions this is difficult-to specify. However, the meaning of *vary* can be understood from the **Solution** aspects of the pattern viz. that new types may be added by subclassing the Observer class independent of the Subject class and types of dependent observers to an object of the Subject class may also be varied at run-time. In summary, though the reuse requirements given in [25] can be stated informally before stating the **Solution**, their meaning can be deduced unambiguously only from the formal specification of the solution since they are stated in terms of the abstraction present in the solution domain of a design pattern. Therefore, in the formal template we state the reuse requirements not as part of **Reuse Requirements** section but as part of the **Solution** section. At the problem domain level, they for most part seem ambiguous. Therefore, we believe that formalism cannot be applied to this section. However, as we explain formal specification of the design pattern solution can help a reuser to deduce how the reuse requirements are met by the solution. In summary, therefore we state that the formalism provided for the **Problem** and **Constraints** completely characterize the problem domain of a design pattern.

5.3.4 Solution Section

The **Solution** section of the informal template 5.2 is identical to the description in [25]. We found several inconsistencies and ambiguities with these descriptions. For instance, the **Reuse Requirements Resolution** section of Abstract Factory pattern

mentions that the class of product objects appears only once in the application viz. where it is instantiated. Thus, product objects used by clients can be varied by configuring the clients with the desired concrete factory. This makes it easier for clients to switch between families. However, the **Participants** section of the **Solution** only states “that clients manipulate product objects through the AbstractFactory’s interface”. Similarly the **Collaborations** section under **Solution** also states that “*normally* a single instance of concrete factory is created at run-time”. There is an inconsistency in the **Solution** in that the solution does not specify how the clients know about or create the product objects. Thus, it is not clear how the product objects are created through the AbstractFactory’s interface, and also why it is easier to interchange families by just choosing a desired factory.

The **Solution** section of Composite pattern uses several ambiguous terms and phrases such as *children of a composite*, *child related operations are implemented in the interface of component* which need to be formally stated. It is also clear from the solution that Composite class is a subtype of the Component class while the Leaf class is a subclass. The collaborations in the **Solution** of Composite pattern are also stated informally. The **Collaborations** section in Observer design pattern states that a ConcreteSubject notifies its observers whenever a change occurs that could make its observer’s state inconsistent with its own. This statement is ambiguous in that it is not clear who is responsible for notifying the observer. The subject can change its state and notify its observers by invoking Notify() on itself. Instead, the responsibility of notifying observers can lie with the client that invokes SetState() on the subject. This ambiguity is intentional; however, the informal statement seems to imply the former scenario even though neither the OMT diagram nor the **Participants** section states this. The contextual example and collaboration sections under the **Solution** clearly spell out that there is a state dependency between such objects. We have found that similar criticisms apply to all patterns found in [25]. Due to space considerations, we do not discuss them.

5.3.5 Reuse Requirements Resolution Section

The **Reuse Requirements Resolution** section describes how the reuse requirements can be met by customizing the solution. It is difficult to deduce them from informal descriptions of the design pattern solution. However, they can be deduced

from the formal representation provided for the **Problem, Constraints and Solution**. Thus we have two levels of abstractions in design patterns (1) the **Problem and Constraints** pertaining to the problem domain of a design pattern; (2) the **Solution, Reuse Requirements and Reuse Requirement Resolution** pertaining to the context of abstractions described in the **Solution**.

5.3.6 Drawbacks Section

Similarly, we do not want to specify the **Drawbacks** aspects formally, since these can also be deduced from the formal representation of the **Problem, Constraints and Solution**. Sometimes they also seem to be derived from the pattern writer's experience and often describe specific instances where the pattern was not applicable. So, we prefer to describe these aspects only informally.

5.3.7 Summary

We believe that a formal approach to documenting patterns can specify the following:

- **Problem** – This is achieved by identifying and specifying the problem domain entities and their characteristics. The specification also specify the entities and their properties that must be provided by the design pattern solution. A specification of the task to be achieved by the pattern may also be provided (if required).
- **Constraints** – This is achieved by specifying the design choices made.
- **Solution** – by specifying the participants, their interfaces and relationships.
- A formal specification of the collaborations in the **Solution**. These specify the collaboration among the objects within the pattern as well the manner in which the clients must configure or collaborate with objects within the pattern.

The rationale behind the design choices made in **Constraints** must not be formally specified. Similarly, the **Reuse Requirements** section can either be informally stated or omitted from the formal template. The **Reuse Requirement Resolution** section can be deduced from the formal representation of the **Solution**. The **Drawbacks** section is also informally stated.

Finally we point out some limitations and caution that formal approaches to design pattern documentation are an extremely tricky and difficult affair. By their very definition, design patterns are intended to have certain ambiguities and vagueness in order to ensure their wide applicability. On the other hand, formal methods are used to bring in certain degree of precision in describing problems. It can be argued that the degree of precision promised by formal methods comes at the risk of sacrificing the high-abstraction level and wide applicability of the pattern. However this is not valid. A design pattern is not merely a design product, but rather a documentation of design knowledge. So, applying formal methods to such documentation only improve the understanding and reuse of the product.

The formal approach that we present identifies the contours of such a formalism and sets the tone for a full-fledged investigation of designing a formal pattern language of the future.

5.4 A Formal Approach

We shall use multisorted first-order logic to specify the abstractions needed. Unlike set-theoretic first order language, this language enables us to specify properties of the sorts. Further, the use of a multi-sorted first order language enables us to specify abstractions which can reuse the traits from the LSL library. We shall also use Larch/C++ style specifications to specify the interface of the classes within a design pattern description. Thus, our notations follow closely the principles of Larch/C++.

5.4.1 Sorts

- **Class:** A class defines an object's interface and implementation. It specifies the object's internal representation and defines the operations the object can perform. The sort `Class` represents the sort of all classes.
- **Signature:** An operation's signature defines its name, parameters, and return value. The sort `Signature` represents the sort of all signatures.
- **Interface:** The set of all signatures defined by an object's operations. Formally, `Set[Signature]` represents the sort of an interface.

- **Method:** The implementation of a signature is referred to as a method. The sort **Method** represents the sort of all such methods.
- **Instance Variables:** A piece of data that defines part of an object's representation is called an instance variable. The sort **Attribute** represents the sort of all such instance variables.
- Instance variables can either represent a relationship (in which case they are references to some class or classes) or they represent a state variable of the object. The sort **StateVar** represents such state variables, and sort **RelationshipVar** represents the relationship variables. The sort **Attribute** represents the sort of all instance variables. Thus we may specify **Attribute**: *tuple* of [Value: Set[StateVar], Reference: Set[RelationshipVar]] (where *tuple* is an abstraction specified in the LSL tier).
- Every instance variable in a class implementing an acquaintance or aggregation relationship establishes a relationship between the owner class and another class with a certain cardinality. The sort **Cardinality** represents the cardinality of such a relationship. We thus have, **Cardinality**: *Enum* of one, one-many, zero-many, at-most one. We also introduce a sort **Token** such that **Token**: *tuple* of [RelatedTo: Class, WithCardinality: Cardinality].
- Since each instance variable representing a relationship in a class is either an acquaintance or an aggregation we introduce sorts **AggregationVar** and **AcquaintanceVar** to represent the aggregation and acquaintance instance variables in a class respectively. We thus have: **RelationshipVar**: *union* of {**AggregationVar**, **AcquaintanceVar**}.
- **Object:** A run-time entity that packages both data and procedures operating on that data. The sort **Object** represents the sort of objects.
- The sort **Obj[Class]** represents the sort of objects that are instances of the parameter class.
- **Type:** A type is simply the name of a particular interface. The sort **Type** represents the sort of types.

Having introduced these basic sorts we introduce several function and predicate symbols on these sorts and also introduce axioms.

5.4.2 Class Level Predicates

Simple Predicates

An abstract class is a class that cannot be instantiated while concrete classes are classes that can be instantiated. We introduce predicates which state that any object of the sort `Class` is either abstract or concrete but not both.

Signatures:

`isAbstract: Class → Bool` – {*check whether a class is abstract*}

`isConcrete: Class → Bool` – {*check whether a class is concrete*}

Axioms:

$\forall c: C \text{ (isAbstract}(c) == \sim \text{isConcrete}(c))$

Class Interface Predicates

In this section we describe properties of a class and its interface. We need notions of abstract operations and class operations: a abstract operation is an operation that defines a signature but does not implement it; a class operation is an operation targeted to a class and not to an individual object.

Signature:

`HasInterface: Class → Interface` – {*every class has a unique interface*}

`∈ : Signature, Class → Bool` – {*check whether the signature is defined by the class' interface*}

`AbstractOp: Signature, Class → Bool` – {*check whether an operation defined in the interface of a class is an abstract operation*}

`ClassOp: Signature, Class → Bool` – {*check whether an operation defined in the interface of a class is a class operation in class.*}

`ImplementsAs: Class, Signature, Method → Bool` {*The class c implements the signature s by method m .*}

Axioms:

$\forall c: C, s: S, m: \text{Method}$

$(s \in c == s \in \text{HasInterface}(c))$

$(\text{AbstractOp}(s, c) \Rightarrow s \in c)$

$(\text{ClassOp}(s, c) \Rightarrow s \in c)$

$\text{ImplementsAs}(c, s, m) \Rightarrow s \in c.$

Class Instance Variable Predicates

In this section we describe certain relations and properties of a class and its attributes.

Signatures:

$\text{HasAttributes}: \text{Class} \rightarrow \text{Attribute} - \{\text{for each class returns its attributes}\}$

$\text{HasType}: \text{StateVar} \rightarrow \text{Class} - \{\text{for each state variable returns the class that represents the type of the state variable}\}$

$\text{OwnedBy}: \text{StateVar} \rightarrow \text{Class} - \{\text{for each state variable returns its owner class}\}$

$\text{OwnedBy}: \text{RelationshipVar} \rightarrow \text{Class} - \{\text{is owned by the class}\}$

$\text{AssociatedWith}: \text{RelationshipVar} \rightarrow \text{Token} - \{\text{instance variable contains reference or references to a class with a certain cardinality.}\}$

$\text{HasRelationship} : \text{Class}, \text{AggregationVar}, \text{Class} \rightarrow \text{Bool}$

$\text{AbstractCoupling}: \text{Class}, \text{Class} \rightarrow \text{Bool}$

Axioms:

$\forall s: \text{StateVar}, c, c1: C, r: \text{RelationshipVar}$

$(\text{OwnedBy}(s) = c == s \in \text{HasAttributes}(c).\text{Value})$

$(\text{OwnedBy}(r) = c == r \in \text{HasAttributes}(c).\text{Reference})$

$\text{HasRelationship}(c, r, c1) == \text{OwnedBy}(r) = c \wedge (\text{AssociatedWith}(r).\text{RelatedTo} = c1)$

$(\text{AbstractCoupling}(c, c1) == \text{isAbstract}(c1) \wedge r \in \text{HasAttributes}(c).\text{Reference} \wedge \text{AssociatedWith}(r).\text{RelatedTo} = c1)$

Class Inheritance Predicates

In this section we introduce the signatures and axioms to describe the inheritance relationships. The *Interface Inheritance* and *Implementation Inheritance* are described by the *InheritsInterface* and *InheritsImplementation* operations.

Signature:

InheritsInterface: Class, Class \rightarrow Bool – {*check whether a class inherits the interface of another class*}

InheritsImplementation: Class, Class \rightarrow Bool – {*check whether a class inherits the implementation of another class*}

InheritsAttributes: Class, Class \rightarrow Bool – {*check whether a class inherits the attributes of another class*}

Parent_Of: Class, Class \rightarrow Bool – {*check whether a class is a parent of another class*}

Subclass_Of: Class, Class \rightarrow Bool – {*check whether a class is a subclass of another class*}

Subtype: Class, Class \rightarrow Bool – {*check whether a class is a subtype of another class*}

Axioms:

$\forall c, c1: C$

$\text{Parent_Of}(c, c1) == \text{Subclass_Of}(c1, c)$

$\text{Parent_Of}(c, c1) == \text{InheritsInterface}(c1, c) \vee \text{InheritsImplementation}(c1, c)$.

$\text{Subclass_Of}(c, c1) == \text{InheritsInterface}(c1, c) \vee \text{InheritsImplementation}(c1, c)$.

$\text{InheritsInterface}(c, c1) \Rightarrow \text{HasInterface}(c1) \subseteq \text{HasInterface}(c)$

$\text{Subtype}(c, c1) == \text{InheritsInterface}(c, c1)$

5.4.3 Object Level Predicates

In this section we describe the notion of an object.

Signatures:

$\text{in}: \text{Object}, \text{Class} \rightarrow \text{Bool} - \{\text{check whether given object is an instance of the given class}\}$

$\text{incarnation}: \text{Object} \rightarrow \text{Class} - \{\text{for each object return the class from where it is instantiated}\}$

$\text{InstanceSet}: \text{C} \rightarrow \text{Set}[\text{Object}] - \{\text{for each class return the the set of instances of that class}\}$

$\text{creation_time}: \text{Object} \rightarrow \text{Nat} - \{\text{denotes the time during the program execution when the object } o \text{ is created}\}$

Axioms:

$\forall o:\text{O}, c, c1: \text{C}, s: \text{Set}[\text{Object}]$

$\text{in}(o,c) == \text{The object is an instance of the class } c.$

$\text{in}(o, c) \wedge \text{in}(o, c1) == c = c1$

$\text{in}(o, c) == \text{incarnation}(o) = c$

$\text{InstanceSet}(c) = s == \forall o: \text{Object} (\text{in}(o,c) \Leftrightarrow o \in s)$

$\text{InstanceSet}(c,s) = \text{InstanceSet}(c1,s) == (c = c1)$

Aggregate Object and Object Reference

In this section we introduce the notions of aggregate object, its parts and object relationships. We assume that any reference stored in a object must be an instance of an acquaintance or aggregation relationship. We are not concerned with how the acquaintance or aggregation relationship is implemented (ie. whether an object stores a reference to another object or a list of references to another object).

Signatures:

$\text{Aggregate}: \text{Object} \rightarrow \text{Bool} - \{\text{an object is aggregate or not}\}$

$\text{Part_Of}: \text{Object}, \text{Object} \rightarrow \text{Bool} - \{\text{an object is part of another.}\}$

$\text{HasReferenceTo}: \text{Object}, \text{RelationshipVar}, \text{Object} \rightarrow \text{Bool} - \{\text{object } o \text{ has a reference to } o1 \text{ and this object reference is an instance of relationship instance variable } r.\}$

$\forall o, o1: \text{Object}, r: \text{RelationshipVar}, c, c1: \text{Class}, a: \text{AggregationVar}$

Axioms:

$\text{Part_Of}(o, o1) \Rightarrow \text{Aggregate_Of}(o1) - \{ \text{if } o \text{ is part of } o1, o1 \text{ must be aggregate object.} \}$

$\text{HasReferenceTo}(o, r, o1) \Rightarrow \text{incarnation}(o) = c \wedge \text{incarnation}(o1) = c1 \wedge \text{HasRelationship}(c, r, c1)$

$\text{HasReferenceTo}(o, a, o1) \Rightarrow \text{Aggregate}(o) \wedge \text{Part_Of}(o, o1)$

5.4.4 Type Predicates

In this section we describe the notion of a type. A type is name denoting an interface, and every class has a unique type associated with its interface.

Signatures:

$\text{Denotes: Interface} \rightarrow \text{Type} - \{ \text{Interface denotes a type} \}$

$\text{HasType: Class} \rightarrow \text{Type}$

Axioms: $\forall i, i1: \text{Interface}, c: \text{C}, t: \text{Type}$

$\text{Denotes}(i) = \text{Denotes}(i1) == i = i1$

$\text{HasType}(c) = t == \text{Denotes}(\text{HasInterface}(c)) = t$

5.4.5 Other Predicates

In this section we introduce predicates that express the notion of object creation, and usage. We assume that there is a distinguished object in the system that creates every other object. The creation of every object in the system can be traced back to this object.

Signatures:

$\text{create: Object, Object} \rightarrow \text{Bool} - \{ \text{object } o \text{ creates the object } o1 \text{ (creation here means that the constructor of the incarnation}(o1) \text{ is called and initialized with } o1) \}$

$\text{CreatesObjectFrom: Object, Class, Class} \rightarrow \text{Bool} - \{ o \text{ is an instance of } c \text{ which instantiates } c1 \text{ by calling new} \}$

$\text{CreatesObjectFrom: Object, Class, Object, Class} \rightarrow \text{Bool} - \{ o \text{ is an instance of } c \text{ and } o \text{ creates } o1 \text{ which is an instance of } c1 \}$

Specifies: Class, Object, Class \rightarrow Bool – *{class c specifies statically that instance variable o1 is created as instance of class c1}*

SpecifiesInitialization: Class, Object, Class, Object, Signature \rightarrow Bool – *{SpecifiesInitialization o1, c1, f, CreateProductA():The class c specifies that the object o1 is an instance variable of class c1 and o1 is constructed by initializing with f.CreateProductA()}*

SpecifiesType:Class, Object, Class \rightarrow Bool – *{class c specifies statically that instance variable o1 is to be used as having type of class c1}*

UsesObjectFrom: Object, Class, Object, Class \rightarrow Bool – *{o is an instance of c and o requests services from o1 which is an instance of c1}*

Axioms:

$\forall o, o1: \text{Object}, c, c1: \text{Class}$

$\text{CreatesObjectFrom}(o, c, c1) \Rightarrow \text{in}(o, c)$

$\text{CreatesObjectFrom}(o, c, o1, c1) \Rightarrow \text{in}(o, c) \wedge \text{in}(o1, c1)$

$\text{Specifies}(c, o1, c1) \wedge \text{Specifies}(c, o1, c2) \Rightarrow c1 == c2$

$\text{SpecifiesInitialization}(c, o1, c1, f, \text{CreateProductA}()) \Rightarrow \text{Specifies}(c, o1, c1)$

$\text{SpecifiesType}(c, o1, c1) \wedge \text{SpecifiesType}(c, o1, c2) \Rightarrow c1 == c2$

5.4.6 Interface and Interaction Specifications

Given, $\text{foo}(\text{ArgType}): \text{Signature}, c, \text{ArgType}: \text{Class}$ such that $\text{foo} \in c$, an interface specification may be provided. Such a specification can either be completely or partially specify the behavior of the interface function foo . We follow Larch/C++ style syntax; however we provide only **requires** and **ensures** clauses. The **modifies** clause is omitted since, design pattern descriptions are highly abstract and objects within such design patterns do not have a data model.

$\text{foo}(o:\text{Argtype})\{\text{requires pre-cond}; \text{ensures post-cond}\}$ where **pre-cond** and **post-cond** are predicates defined over the vocabulary of LSL, primitive predicates introduced and predicates introduced in the specification of each design pattern. The semantics is that the before the execution of the operation foo , the pre-condition must be true and after the execution the post-condition holds.

For each class within the pattern whose objects collaborate with other objects in the pattern by method delegation, we use the following notation to describe this collaboration:

ClassName Package:

Operation1():*Collaboration_Specification1*

Operation2():*Collaboration_Specification2*

End Package

The semantics is that the listed operations in the package result in a collaboration as specified by the collaboration specification following the colon. Only those operations that participate in a collaboration are listed in the package. Note, that the design patterns descriptions in [25] only provide pseudocode and at times object interaction graphs [6] to describe the collaborations. Also, the description of the interface operations are incomplete and only the method delegations pertinent to the pattern are provided. For instance, in a pattern such as Adapter, the Adapter class forwards requests to the Adaptee, but it may provide some behavior before or after forwarding this request. Due to these reasons, we cannot use the collaboration specification introduced in Chapter 4. For, the purposes of describing the collaborations within pattern in [25], it seems sufficient to provide notations to capture the method delegations and sequential composition. We therefore introduce the following operators:

- **o.Operation:** This indicates the invocation of Operation on object o. When the object on which Operation is invoked is clear from the context we only write Operation instead of o.Operation.
- **A(o): o.Operation:** This indicates the invocation of Operation on all objects that satisfy the predicate A(o).
- **P;Q:** This indicates the sequential composition of P and Q where P and Q are the operators specified earlier.

Thus, the expression **Operation:***CollaborationSpecification* in the package specification given above has the semantics that the invocation of the Operation on an object of the Class (whose name is provided by the package name) results in a series of invocations or method delegations as defined by the above operators.

Similarly, in design pattern descriptions we often encounter constraints that a client must respect in configuring the objects in the pattern or in requesting a service from an object in the pattern. In such cases, we have to specify the order in which client requests services from objects in the pattern. For instance, in the Observer design pattern 5.2.3, a client is expected to follow up a SetState request on an instance of Subject with an Update on all its dependents (that are instances of the Observer). In the examples from [25] that we considered we have felt the need for the following constraints:

Collaboration Constraints: Given $o1:Obj[C1]$, $o2:Obj[C2]$: $\{o1.Operation1; o2.Operation2\}$

The semantics is that if o and $o1$ are instances of $C1$ and $C2$ respectively, then the invocation of $Operation1$ on $o1$ is always followed by the invocation of $Operation2$ on $o2$.

Collaboration Constraints: Given $o1:Obj[C1]$, $\forall o2: Obj[C2]$: $A(o) \{o1.Operation1; o2.Operation2\}$

The semantics is that if $o1$ is an instance of $C1$ and $o2$ is an instance of $C2$ satisfying the given predicate $A(o)$ then the invocation of $Operation1$ on $o1$ is always followed by the invocation of $Operation2$ on $o2$.

5.5 Applying the Formalism

We have chosen three design patterns from [25] to apply the formalism discussed in the section 5.4. Our experience suggests that the formalism is sufficiently applicable to other design patterns described in [25] as well. The patterns chosen to apply our formalism are Abstract Factory, Composite and Observer. We shall follow the following template and the reader may refer to the figures provided earlier:

5.5.1 Formal Template

- **Name:** Represents the name of the pattern.
- **Problem:** A brief informal description of the problem.

Example Problem:

Formal Representation:

- **Constraints:** Informal description of design constraints (as explained above).
Example Constraints:
Constraint Formalism:
- **Reuse Requirements:** Informal description of the reuse requirements.
Example Reuse Requirements:
- **Solution:** A formal description of the solution. These may be supplemented with informal descriptions as required.
Structure:
Collaborations:
Example Solution:
- **Reuse Requirements Resolution:** Description of the reuse requirements and how they are resolved. These requirements are resolved by the **Solution** subject to the fact that they meet the specification of the **Problem** and **Constraints**.
- **Drawbacks:** The negative consequences of using the design pattern.
- **Implementation issues and Sample Code.**

Since, the reuse requirements stated in **Reuse Requirements** section are repeated in **Reuse Requirement Resolution** section (except that the latter also explains how they are resolved by the design pattern solution) and we have already explained that the reuse requirements are only understood in the context of abstractions of **Solution**, we suggest that **Reuse Requirements** section be optional. In the following examples we omit the **Reuse Requirement** section. We also do not annotate the section with example or figures and refer the reader to earlier sections for the same. Similarly we have also omitted providing informal description of the **Constraints**, **Solution** and **Collaboration** but these can supplement the formal description as indicated in the above formal template. We refer the reader to the informal descriptions of the patterns provided in 5.2.1, 5.2.2 and 5.2.3.

We present below the Abstract Factory, Composite and Observer design patterns [25].

5.6 Abstract Factory

In the formal approach given below for Abstract Factory design pattern, we identify certain classes in the problem domain, and describe their properties. For instance, we specify that the problem domain has certain product objects that have the same type. The relationships between such objects is also made clear. The constraint is also specified in terms of the product objects specified earlier. Note that we also specify as part of the problem the requirement that clients use objects from only one family. In [25] this is stated only as part of the contextual example. We believe that this is part of the design issue discussed in the **Problem** and must therefore be specified as part of the **Formal Representation**. The specifications of Composite and Observer design patterns follow similar principles.

5.6.1 Problem

Clients want to create families of product objects.

Formal Representation

- ProductA1, ProductA2, ProductB1, ProductB2 : Class
- Signatures:
 - InFamily1: Object \rightarrow Bool – {*check whether an object is in family 1.*}
 - InFamily2: Object \rightarrow Bool – {*check whether an object is in family 2.*}
 - InSameFamily: Object, Object \rightarrow Bool – {*check whether objects are in the same family*}
 - InEitherFamily: Object \rightarrow Bool – {*check whether objects are in one of the families*}
- Axioms: $\forall o, o1, o2, o3$: Object
 - HasInterface(ProductA1) == HasInterface(ProductA2)
 - HasInterface(ProductB1) == HasInterface(ProductB2)
 - InFamily1(o) == in(o, ProductA1) \vee in(o, ProductB1)
 - InFamily2(o) == in(o, ProductA2) \vee in(o, ProductB2)

- $\text{InSameFamily}(o, o1) == \text{InFamily1}(o) \wedge \text{InFamily1}(o1) \vee \text{InFamily2}(o) \wedge \text{InFamily2}(o1)$
- $\text{InEitherFamily}(o) == \text{InFamily1}(o) \vee \text{InFamily2}(o)$
- $\text{create}(o, o1) \wedge \text{create}(o2, o3) \wedge \text{InEitherFamily}(o1) \wedge \text{InEitherFamily}(o3) \Rightarrow \text{InSameFamily}(o1, o3)) - \{ \textit{The client objects create product objects from the same family.} \}$

5.6.2 Constraints

Constraint Formalism $\forall o, o1: \text{Object}, c: \text{Class}$

- $\sim (\text{CreatesObjectFrom}(o, c, o1, \text{ProductA1}) \wedge \text{Specifies}(c, o1, \text{ProductA1}))$
- $\sim (\text{UsesObjectFrom}(o, c, o1, \text{ProductA1}) \wedge \text{SpecifiesType}(c, o1, \text{ProductA1}))$

Similar constraints hold for instances of classes ProductA2, ProductB1, and ProductB2.

The above assertion expresses the constraint that the client does not specify statically the class of product objects it will create at run-time. The rationale is that, specifying the class of products explicitly would make it difficult to switch between products both statically as well as dynamically.

5.6.3 Solution

Structure

- AbstractFactory, ConcreteFactory1, ConcreteFactory2, ProductA, ProductA1, ProductA2, ProductB1, ProductB2: Class, factory: Object
- CreateProductA, CreateProductB : Signature
- {return new ProductA}, {return new ProductB} : Method
- isAbstract(AbstractFactory)
- isAbstract(ProductA)

- `isAbstract(ProductB)`
- `CreateProductA() ∈ AbstractFactory`
- `CreateProductB() ∈ AbstractFactory`
- `ImplementsAs(AbstractFactory, CreateProductA(), {return new ProductA})`
- `ImplementsAs(AbstractFactory, CreateProductB(), {return new ProductB})`
- `Subtype(ConcreteFactory1, AbstractFactory)`
- `ImplementsAs(ConcreteFactory1, CreateProductA(), {return new ProductA1})`
- `ImplementsAs(ConcreteFactory1, CreateProductB(), {return new ProductB1})`
- `Subtype(ConcreteFactory2, AbstractFactory)`
- `ImplementsAs(ConcreteFactory2, CreateProductA(), {return new ProductA2})`
- `ImplementsAs(ConcreteFactory2, CreateProductB(), {return new ProductB2})`
- `Subtype(ProductA1, ProductA)`
- `Subtype(ProductA2, ProductA)`
- `Subtype(ProductB1, ProductB)`
- `Subtype(ProductB2, ProductB)`
- The clients satisfy the following:
 - $\forall o, o1: \text{Object } c: \text{Class } (\text{CreateObjectFrom}(o, c, o1, \text{ProductA1}) \Rightarrow \text{SpecifiesInitialization}(c, o1, \text{ProductA}, f, \text{CreateProductA}()))$
 - $\forall o, o1: \text{Object } c: \text{Class } (\text{UsesObjectFrom}(o, c, o1, \text{ProductA1}) \Rightarrow \text{SpecifiesType}(c, o1, \text{ProductA}))$

Collaborations

- Client initializes the factory variable *factory* statically to an instance of class `ConcreteFactoryi` ($i = 1$ or 2). In this case: $\forall o: \text{Object } (\text{InEitherFamily}(o) \Rightarrow \text{creation_time}(f) < \text{creation_time}(o))$

- Client initializes the factory variable *factory* dynamically to an instance of the desired factory `ConcreteFactoryi` ($i = 1$ or 2). In this case: $\forall o:\text{Object} (\text{InEitherFamily}(o) \Rightarrow \text{creation_time}(f) < \text{creation_time}(o))$

5.6.4 Reuse Requirements

- Clients can easily vary the family of products being used. This is clear from cases given above under **Collaborations**. It is clear that in either case the client requires little or no modification. The clients can easily switch between families.
- Easy to add new families of products: To add new products that satisfy the constraints specified in the problem, we just have to subclass certain classes in the Abstract Factory pattern.

5.7 Composite

5.7.1 Problem

Compose objects into tree structures to represent part-whole hierarchies.

Formal Representation

- Primitive: `Class, Operation()` \in Primitive.
 - Define Composite: Class such that
 - $\forall o: \text{Object} (\text{in}(o, \text{Composite}) \Rightarrow \text{Aggregate}(o) \wedge (\forall o1: \text{Object} (\text{Part_Of}(o1, o) \Rightarrow \text{in}(o, \text{Composite}) \vee \text{in}(o, \text{Primitive}))))$
 - Further, define: `Add(Primitive) \in Composite`, `Add(Composite) \in Composite`, `Remove(Primitive) \in Composite`, `Remove(Composite) \in Composite`, and `Operation() \in Composite` such that:
 - `Add(p: Primitive){ensures Part_Of(p, self)}`
 - `Remove(p: Primitive){ensures Part_Of(p, self)}`.
- Composite Package:
- `Operation(): $\forall o: \text{Object} (\text{Part_Of}(o, \text{self}): o.\text{Operation}()$`

5.7.2 Constraints

Constraint Formalism

$\forall o, o1, o2, o3: \text{Object} \forall c, c1, c2: \text{Class} (\text{UsesObjectFrom}(o, c1, o1, c2) \wedge (c2 == \text{Primitive}) \vee (c2 == \text{Composite}) \Rightarrow \text{SpecifiesType}(c1, o1, c))$

The above constraint states that the client uses objects from the Primitive and Composite classes uniformly (ie. as being of the same type).

The rationale is that the clients use primitive and composite objects the same way. So treating them differently would make the application unnecessarily complex.

5.7.3 Solution

Structure

- Component, Primitive, Composite: Class
- $\{\forall o: \text{children } o.\text{Operation}\}$: Method
- isAbstract(Component)
- isConcrete(Primitive)
- isConcrete(Composite)
- Add(Component) \in Component
- Remove(Component) \in Component
- Operation() \in Component
- Subtype(Composite, Component)
- Subclass(Primitive, Component)
- Operation() \in Primitive
- children: AggregationVar, OwnedBy(children, Composite), Aggregate_Of(children, [Component, one-many]), Aggregate_Of(Composite, children, Component)

- Composite: $\text{Add}(o:\text{Component})\{\text{requires } \forall c:\text{Class } \text{Subclass}(c, \text{Component}) \wedge \text{in}(o, c); \text{ensures } \text{HasReferenceTo}(\text{self}, \text{children}, o)\}$
- Composite: $\text{Remove}(o:\text{Component})\{\text{requires } \forall c:\text{Class } \text{Subclass}(c, \text{Component}) \wedge \text{in}(o, c); \text{ensures } \sim\text{HasReferenceTo}(\text{self}, \text{children}, o)\}$
- Implements(Composite, Operation, $\{\forall o: \text{children } o.\text{Operation}\}$)
- $\forall o, o1: \text{Object } \text{UsesObjectFrom}(o, o1, \text{Primitive}) \vee \text{UsesObjectFrom}(o, o1, \text{Composite}) \Rightarrow \text{SpecifiesType}(o, o1, \text{Component})$

The last assertion states that all clients use the interface of Component class to manipulate the primitive and composite objects.

Collaborations:

Composite Package:

Operation() : $\forall o: \text{Object } \text{HasReferenceTo}(\text{self}, \text{children}, o) o.\text{Operation}()$

End Package

5.7.4 Reuse Requirements

- Easy to add new types of Primitive and Composite objects. This can be done by subclassing Component and Composite classes in the design pattern.
- Easy to vary the structure and composition of the Composite object. This is clear from the requires clause of Add(Component) and Remove(Component) operations in the interface of Composite class.

5.8 Observer

5.8.1 Problem

Define a one-to-many dependency between objects such that when one object changes state all its dependents are notified and updated automatically.

Formal Representation

- Subject, Observer: Class

- Dependency: $\text{Obj}[\text{Subject}] \rightarrow \text{Set}[\text{Obj}[\text{Observer}]] - \{1\text{-many dependency between Subject and Observer.}\}$
- Consistency: $\text{Obj}[\text{Subject}] \rightarrow \text{Obj}[\text{Observer}] - \{state consistency between objects of Subject and Observer classes.}\}$
- $\text{SetState}() \in \text{Subject}, \text{SetState}\{\text{ensures } \forall o: \text{Obj}[\text{Observer}] o \in \text{Dependency}(\text{self}) \Rightarrow \sim \text{Consistent}(\text{self}, o)\}$
- $\text{Update}() \in \text{Observer}, \text{Update}()\{\text{ensures } \forall s: \text{Obj}[\text{Subject}] (\text{self} \in \text{Dependency}(s) \Rightarrow \text{Consistency}(s, \text{self})\}$
- Given $o: \text{Obj}[\text{Subject}] \forall o1: \text{Obj}[\text{Observer}] o1 \in \text{Dependency}(o)\{o.\text{SetState}; o1.\text{Update}\}$.

The collaboration constraint states that clients must follow up a SetState message on an object of the Subject class with an Update message on all its dependent objects.

5.8.2 Constraint Formalism

$\forall o: \text{Obj}[\text{Subject}], o1: \text{Obj}[\text{Observer}] (o1 \in \text{Dependency}(o) \Rightarrow \sim(\exists r: \text{RelationshipVar} (\text{RelatedTo}(\text{Subject}, r, \text{Observer}) \wedge \text{HasReferenceTo}(o, r, o1))))$

The constraint states that the Subject and Observer classes should not be tightly coupled. This means that the Subject class must not store references to the Observer class, yet we want the subject to communicate with its dependent observers.

5.8.3 Solution

Structure

- AbstractSubject, Subject, AbstractObserver, Observer: Class
- $\{\forall o: \text{observers } o.\text{Update}\}$
- isAbstract(AbstractSubject)
- isAbstract(AbstractObserver)

- isConcrete(Subject)
- isConcrete(Observer)
- observers:AcquaintanceVar, OwnedBy(observers, AbstractSubject), Acquaintance_Of([AbstractObserver, one-many]), Acquaintance_Of(AbstractSubject, observers, AbstractObserver)
- subjectstate: StateVar, OwnedBy(subjectstate, Subject)
- Attach(Observer) ∈ AbstractSubject
- AbstractSubject:Attach(o:Observer){ requires $\forall c: \text{Class Subclass}(c, \text{Component}) \wedge \text{in}(o,c)$; ensures $\text{HasReferenceTo}(\text{self}, \text{observers}, o)$ }
- Detach(Observer) ∈ AbstractSubject
- AbstractSubject:Detach(o:Observer){ requires $\forall c: \text{Class Subclass}(c, \text{Component}) \wedge \text{in}(o,c) \wedge \text{HasReferenceTo}(\text{self}, \text{observers}, o)$; ensures $\sim \text{HasReferenceTo}(\text{self}, \text{observers}, o)$ }
- AbstractSubject:Detach(o:Observer){ ensures $\sim \text{HasReferenceTo}(\text{self}, \text{observers}, o)$ }
- Notify() ∈ AbstractSubject
- Implements(AbstractSubject, Notify, { $\forall o:\text{observers } o.\text{Update}$ })
- Subtype(Subject, AbstractSubject)
- SetState ∈ Subject
- Subject:SetState {ensures $\forall o: \text{Obj}[\text{Observer}] (\text{HasReferenceTo}(\text{self}, \text{observers}, o) \Rightarrow \sim \text{Consistent}(\text{self}, o))$ }
- Subject:GetState {ensures return subjectstate}
- Update() ∈ AbstractObserver
- Subtype(Observer, AbstractObserver)

- $\text{subject:AcquaintanceVar, OwnedBy}(\text{subject, Observer}), \text{Acquaintance_Of}(\text{subject, [Subject, zero-one]}, \text{Acquaintance_Of}(\text{Observer, subject, Subject}))$
- $\text{Observer:Update}\{\text{ensures } \forall s:\text{Obj}[\text{Subject}] (\text{HasReferenceTo}(\text{self, subject, s}) \Rightarrow \text{Consistency}(s, \text{self}))\}$

Collaborations:

Subject Package:

$\text{Notify}() : \forall o:\text{Object } \text{HasReferenceTo}(\text{self, observers, o}) o.\text{Update}$

End Package

Observer Package:

$\text{Update}(): \forall o:\text{Object } \text{HasReferenceTo}(\text{self, subject, o}) o.\text{GetState}()$

End Package

Collaboration Constraint: Given $o:\text{Obj}[\text{Subject}] \{o.\text{SetState}(); o.\text{Notify}()\}$

5.8.4 Reuse Requirements

- Easy to vary the number of observers: This is clear from the ensures clause of the $\text{Attach}(\text{Observer})$ and $\text{Detach}(\text{Observer})$ functions in the interface of Subject.
- Easy to vary the type of observers: This is clear from the requires clause in the specification of the $\text{Attach}(\text{Observer})$ in the interface of Subject class.

Chapter 6

Design of a Pattern Repository

In this chapter we propose an object-oriented database schema for representing design patterns and retrieving meaningful information from design patterns towards their reuse. A design pattern does not just represent a design, but also *documents* the intent, rationale and consequences of applying the design over various contexts. A design pattern is indeed a document with many components, each describing some information about the design issue addressed by the pattern and the solution it proposes. We therefore, will use the terms *design pattern documentation* and *design pattern* synonymously throughout this chapter.

6.1 Motivation

The motivation for studying the above problem stems from the following:

- The design pattern community has witnessed a proliferation of design patterns over the last few years. Existing applications are also being mined for design patterns and we can expect large volume of patterns in the future. GOF's design pattern catalog [25], and Ward Cunningham's pattern repository are some of the popular design pattern catalogs. Recently, hundreds of design patterns have been made public by Siemens Ltd.
- As explained in earlier Chapters, design pattern catalogs tend to follow varying documentation styles. The documentation tends to be highly abstract and verbose. Manually browsing through pattern documentations to select a pattern for reuse can be an arduous and time consuming task. For instance, Gamma

et al. [25] provide a broad set of guidelines for selecting a pattern from their catalog. One such guideline is scanning the *Intent* sections of all the patterns in the catalog. However the highly abstract nature of the *Intent* may not help the reuser understand the pattern or decide whether the pattern is suitable for his/her problem. A large catalog documented in this style may compound the task of identifying the right pattern.

- Experienced designers often have partial specification of the desired design structure. Manual catalogs do not distinguish between novice and experienced users and provide no faster means for retrieving design patterns for such users.
- There is much interest in the software reuse community in the development of tools and environments that would support large scale design reuse. The Aesop system [49] developed at Carnegie Mellon University is an experimental platform that minimizes the cost of building systems by providing a generic infrastructure of common tools (design database, GUI, editors, protocol consistency checkers, Software Shelf etc.). This system uses a repository of design elements called Software Shelf that supports the classification, storage and retrieval of architectural elements. However the architectural elements considered are high-level architectural abstractions (such as pipes, event broadcast and other complex protocols). The Software Shelf does not support the representation or retrieval of implementation level design patterns found in [25].
- The *DaVinci* initiative at Massachusetts Institute of Technology seeks to develop an integrated reuse environment with a database of design patterns as a back-end. The goal is to support a collaborative development of software applications by a group of software specialists from a library of software components. However, the project is still at the proposal stage and details about its status are not available.

There is thus a strong motivation and need for designing a repository of design patterns to support their large scale reuse.

6.2 Problem

Our goal is to design a database for representing design patterns and to efficiently query information from such patterns pertinent to their reuse. A first step towards supporting large scale reuse of design patterns is for pattern users to agree on a uniform methodology for documenting them. In Chapter 5, we proposed a uniform methodology for documenting patterns. We shall use the same documentation style as a basis for representing design patterns in our database. Moreover, we shall concern ourselves with only the abstract representation of design patterns in our database. We do not represent concrete instantiations of a design pattern or implementation issues concerning design patterns in our database.

Database modelling of design patterns is a challenging problem. We shall discuss some of these challenges in detail in later sections. For now, we remark that database technology in general is relevant for structured objects. Conventional database applications deal with highly structured information that are expressible using a formal model. Design pattern documentations are composed of various components, but each of these components contains information that does not have pre-defined structure in a flat (eg. relational) or nested (eg. object-oriented) manner. Some of the primary reasons are, the wide range of design issues that each problem addresses, the kinds of subjective information they contain, and very dynamic nature of software industry (that affects both the intent and extent of the proposed pattern repository). Besides, different companies may maintain their own repositories and train their employees in the use of such repositories. Although standardizing pattern documentations styles and type of information they contain is desired, it does seem a difficult thing to realise. In spite of these difficulties inherent in designing of a repository for patterns we believe that recent developments in database technology can be useful in the design of such a repository.

The design of the given database is handled based on requirement analysis. Such requirements can be expressed by a set of queries that potential users of future database can address. In the following we provide a set of queries that will most likely be addressed by the design pattern users to understand the content of the repository and to retrieve relevant information. We note that we shall be addressing these queries only to the abstract representations of patterns and not to their specific instantiations. The rationale is that Gamma et al. [25] provide specific examples

of patterns to motivate the context of a pattern and illustrate its use to aid in the understanding of the abstract pattern. But, the reuser must understand the abstract pattern representation to reuse in various contexts. Also, from the database viewpoint, though the concrete instantiations can be stored passively, querying cannot be done on them since the instantiations are provided in more than one implementation language and it is not possible to come up with a schema to capture the instantiation specific information.

We classify the possible set of queries that can be addressed to such a database into queries that extract information from a specific design pattern (*Intra Design Pattern Queries*) and queries which retrieve a set of design patterns from the repository with certain common properties (*Inter Design Pattern Queries*).

6.2.1 Inter Design Pattern Queries

These class of queries help the designer filter the design patterns in the repository that are most likely related to design needs.

- What are the design patterns that address the given design issue?
- What are the design patterns that have the given design constraint?
- What are all the design patterns that deal with the given design issue and which consider the given design constraint?
- What are all the design patterns that have a *Class* scope?
- What are all the design patterns that have an *Object* scope?
- What are all the design patterns whose purpose is creational?
Similar queries can be posed for structural and behavioral patterns.
- What are all the design patterns that can be used in the given domain and in what context within that domain?
- What are the design patterns used in designing the given framework?
- What are all the relationships between the given two design patterns?
- What are all the design patterns that are related under the given relationship?

- What are the design patterns that are close to a partial design description supplied by the designer?

Such a query is a generic conjunctive request on parts of a design pattern description.

- What are the design patterns for which solution structure contains one or more of the following structures
 - the given class
 - the given class with the interface (provide a partial description of the interface)
 - the structure where class A inherits class B or a structure with the given inheritance hierarchy
 - the structure where class A has an acquaintance relationship with class B.
 - the structure where class A is aggregate of class B.
 - the structure where class B overrides the given signature in its superclass A.
 - the structure where a class A implements one of its signature as a template method
 - the structure described as a conjunction of the above queries (for instance, user can query to retrieve design patterns whose solution contains a recursive aggregate structure or template methods etc).
 - the structure where class A instantiates class B in the given signature.
 - the structure where class A delegates request m to operation m1 in an object o1 (that is an instance variable/argument or local variable to m) of class B.
 - the object structure where aClassA (denoting an arbitrary object of class A) delegates request m1 to request m2 of aClassB which delegates request to ...
 - the object structure where a client requests m1 from aClassA followed with ...

6.2.2 Intra Design Pattern Queries

These set of queries aim to learn more about a given design pattern.

- What are the design constraints in the given pattern and what is their rationale?
- What is the rationale behind a given design constraint in the given pattern?
- What are all the entities in the given design pattern solution (DPS)?
- What are the entities in a given design pattern solution that implement the problem domain entities?
- What is the responsibility of a given class in the given design pattern solution?
- What is the interface and behavior description of the given class in the given design pattern solution?
- What is the static configuration between a client and classes in the given design pattern solution?
- What is the behavior description of the given operation in the given class of the given design pattern solution?
- What are the static associations in a given design? (these include inheritance hierarchies, aggregation and inheritance relationships)
- What is the nature of collaboration between a client and objects in the design pattern solution? (what services are required by the client at run-time and how they are handled - typically these are the services mentioned in the problem statement)
- What is the nature of collaboration between objects in the design pattern solution? (in what order, services collaborate to achieve the task mentioned in the problem statement)
- What are the reuse requirements met by the given design pattern solution?
- What aspects can vary in the design pattern and why?
- How can a certain aspect vary without varying some other aspect in the given design pattern solution?

- Can a certain aspect vary independent of another aspect in the given design pattern?
- What are the design patterns that are related to a given design pattern?
- What are the patterns that are related to a given design pattern and what is the nature of this relationship?

6.3 Choosing a Database Management System

Object-oriented database methodologies, tools and techniques can offer a set of facilities that may be very useful and efficient for the design and implementation of the design pattern (DP) repository.

Besides having the potential of databases in general (e.g., nonprocedural querying, query optimization and processing, schema evolution), OODB technology offers features inherent to the OO paradigm (class, object, object identity, class hierarchy, methods, encapsulation, polymorphism) and facilities provided by OO design and analysis methodologies such as semantic expressiveness, simplicity, modularity and extensibility.

The motivations that lead us to use the OODB technology are:

- The complex structure of design pattern documentation,
- the natural way to express and extend the structural and behavioral description of DPs using OO design tools,
- the facilities offered by database systems in storing and retrieving large amounts of data.

Despite the fact that DP are by essence non-structured (both in terms of their intent and extent) and verbose, we believe that advanced database technology can be a good framework to describe and manage DP documentation. By advanced database technology we mean DB technology with OO principles, deduction mechanisms, induction facilities, information indexing and retrieval, or any variant of these extensions.

We thus use object-oriented techniques (OMT) [52] to design the conceptual schema of our database and also recommend the use of an object-oriented database

management system (OODBMS) to implement the schema. Object models in OMT permit developers to think about a problem at a high, abstract level and yet assure the resulting design to be easily and practically implemented. Object-oriented concepts such as encapsulation and inheritance allow attributes (ie. database design) and programs to be reused as the basis for building complex databases and programs. They thus reduce the difficulty of developing and evolving complex software systems and designs.

An OODBMS unites two technologies: database management and object-oriented programming. Object-oriented programming languages are expressive but lack data persistence (data that outlasts the execution of a single job). Conventional database management systems (DBMS) have data persistence but lack expressibility. OODBMSs try to provide both data persistence and expressiveness. There are a number of commercial object-oriented database management systems such as GemStone from GemStone Systems Inc., O2 from O2 Technology (France), Object Store from Object Design, Inc. These products all support an object-oriented data model. Specifically, they allow the user to create a new class with attributes and methods, have the class inherit attributes and methods from superclasses, create instances of the class each with a unique identifier, retrieve the instances either individually or collectively, and load and run methods. One key objective and, therefore the selling point, of most of the recent OODBMSs is the support of a unified programming and database language; that is, one language (eg. C++ or Smalltalk) in which to do both general purpose programming and database management. A general purpose programming language and a database language are very different in syntax and datamodel (data structures and data types), and the necessity of having to learn and use two very different languages to write database application programs has been frequently regarded as a nuisance.

There is a wide spread belief among the object-oriented database community that OODBMSs can bring about a significant jump in the productivity of database application programmers and even in the performance of application programs [65]. One source of this productivity jump is the reuse of a database design and programs that object-oriented concepts make possible for the first time in the evolving history of database technologies. Another source of significant improvement is the powerful data type facilities implicit in the object-oriented concepts of encapsulation and inheritance. These facilities eliminate three of the most important deficiencies of relational

database management systems (RDBs) [65]. These are:

- RDBs force the user to represent hierarchical data (or complex nested data or compound data) such as a bill of materials in terms of tuples in multiple relations. In addition to a cumbersome approach to modelling data, to retrieve data spread out in multiple relations, RDBs must resort to joins which are relatively expensive operations. The data type of an attribute of an object in object-oriented programming languages (OOPs) can be a primitive type or an arbitrary user defined type (class). The fact that an object has an attribute whose value can be another object naturally leads to a nested object representation, which in turn allows hierarchical data to be naturally (ie. hierarchically) represented.
- RDBs offer a set of primitive, built-in data types for use as domains of columns of relations but do not offer any means of adding user-defined data types. The built-in data types are basically all number and short symbols. RDBs are not designed to allow new data types to be added and, therefore often require a major surgery to the system architecture and code to add a new data type. Adding a new data type to the database system means allowing its use as the data type of an attribute; that is, allowing storage of data of that type, querying and updating such data. Object encapsulation in OOPs does not impose any restriction on the types of data that the data part of an object can hold; that is, the types of data can be primitive types or user defined types. Further, new data types can be treated as new classes, possibly even as subclasses of existing classes, inheriting their attributes and methods.
- Stored procedures in RDBs are not encapsulated with data; that is they are not associated with any relation or any tuple of a relation. And, since RDBs do not have the inheritance mechanism, stored procedures cannot be automatically reused. Object encapsulation is the basis for storage and management of programs as well as data in the database.

In general there are two types of database queries: set-oriented and navigational. RDBs are intended to perform parallel operations on large sets of data. In contrast, OODBMS manipulation languages are efficient at quickly navigating from one object to another by traversing pointers. A RDB performs navigation by using joins, which

are several orders of magnitude slower than pointer traversal. An important feature of OODBMSs is the implicit assumption that the system is oriented towards operations on individual objects and the programmer can expect these to perform well. This is notable mainly because the RDBs typically perform badly for single-object operations and navigation between objects [10, 46].

It is clear from our documentation style, that we can consider each design pattern as a complex object that has a nested structure consisting of components that represent the Problem, Constraints, Solution, Reuse Requirements, and Drawbacks aspects of the pattern. It is also clear that modelling entities in the Solution aspects of design pattern requires meta-modelling techniques since entities to be modelled are themselves classes, attributes, methods, inheritance, aggregation and acquaintance relationships, object collaborations etc. We therefore believe that the expressive power of object-oriented modelling techniques and OODBMSs and the querying mechanism they provide would be very helpful in the design of such a repository. We do note that OODBMSs do have many limitations and still lack many of the features found in RDBs such as a full non-procedural query language, automatic query optimization and processing, automatic concurrency control, authorization, dynamic schema changes and parametrized performance tuning [65].

6.4 Towards the Design

In general there are two approaches to database design [52]. The first approach (bottom up) is attribute driven: Compile a list of attributes relevant to the application and synthesize groups of attributes that preserve the functional dependencies. The other approach is entity driven: Discover entities that are meaningful to the application and describe them. In a typical design there are ten times fewer entities than attributes, so entity design is much more tractable. Object modelling (OMT) [52] is a form of entity design. We shall confine ourselves to designing the conceptual schema of our database using OMT techniques. The conceptual schema is a high-level description of the structure of the database, independent of the particular OODBMS software that will be used to implement the database. The focus in the conceptual schema is essentially on the logical structure of data within the repository, their abstract representation and relationships without concern for efficiency and implementation

issues. The object model used to describe the conceptual schema is also called as conceptual model. In databases designed using OMT techniques the object model forms the most important part of the development life cycle. We thus consider our work to be good step in the direction of full fledged design and implementation of this repository in OODBMS.

6.5 The Conceptual Schema

In this section we present the conceptual schema of our database. The conceptual schema is motivated by the query set provided in sections 6.2.1, 6.2.2 and the discussion in section 6.2. We also confine our discussion to patterns found in Gamma et al [25]. Figures 14 and 15 provide an object model of the schema. In order to preserve clarity in the figures, we have not provided details on representation of the objects and relationships. For these we refer the reader to the next section.

Since the queries are posed over either a collection of patterns or a specific design pattern we represent each design pattern as a nested object with components Problem, Solution, Drawbacks (Please refer to Figure 14). The class Meta Design Pattern models design pattern objects. The Meta Design Pattern class has therefore an aggregation relationship with the classes that model the Problem, Solution and Drawbacks of a design pattern object. Each instance of the Meta Design Pattern class is related to one or more instances of Meta Design Pattern class in one or more ways. For instance, the Decorator pattern is often used with a Composite pattern and their structures are also related (a Decorator can be viewed as a degenerate composite with only one component [25]). Similarly, a design pattern may be used to implement another. For instance, the Factory Method pattern is often used to implement CreateProduct methods in the Abstract Factory pattern, a Builder pattern may be used to construct Composite objects in the Composite pattern and so on. Besides, new pattern relationships can be discovered by designers who use them. Thus, we associate with each relationship between patterns, an attribute that represents the various ways in which they are related. Since, it is not possible to enumerate all possible ways in which patterns may be related we represent this attribute as a set of strings. Each instance of Meta Design Pattern class further has a name, aliases, its scope, purpose [25], a set of tuples where each tuple consists of a problem domain (such as GUI, Distributed

Processing, Relational Database Design) and a set of contexts within that domain where the pattern is used, and a similar set of tuples that represent the frameworks and contexts within such framework where the pattern is used.

Each instance of the Problem class further consists of Core Problem and Constraints which model the design problem addressed by the design pattern and the constraints associated with each design pattern (these have been discussed in detail in Chapter 5). For reasons given in earlier sections we cannot have a formal model for representing the problem and constraints associated with a design pattern. In Chapter 5 we have provided a formal specification of certain aspects of pattern documentation but this formal specification does not provide a formal representation. Ideally we would like the user to provide a specification of the design issue in terms of the language introduced in Chapter 5 and the query engine can map the specification to a specification in the database schema. Some possible extensions of this work will provide this facility with deductive capabilities. In this thesis, we represent the problem and constraints by means of a set of key words and full text. Key words will help in quick retrieval of design patterns that have common key words, and full text description will enable the reuser to learn more details about design pattern attributes. We require our repository to provide full-text search capabilities to query patterns based on their problem or constraints. These key words are selected so that they best represent the intent of the design pattern. We do not elaborate on this aspect as there is currently lot of progress in the area of information retrieval. Besides the key words meant to provide a full-text search facility every instance of the Constraint is a tuple of a design choice made in the pattern and the rationale for the design choice. The Drawbacks component of a design pattern represents the drawbacks of using the pattern and this is represented as text. We believe that the user can be provided with information retrieval tools for indexing and retrieving relevant data. Since our focus is essentially on querying aspects that are crucial to reuse ,viz. retrieval and understanding of a design pattern, we have not tried to elaborate more on this aspect.

Figure 15 provides a high-level description of the Solution schema. We now describe the conceptual schema associated with each design pattern solution. A design pattern solution represents an object-oriented design. It therefore has a static structure as well as certain dynamic properties described in terms of collaboration between the objects. Further, the design pattern solution is a reusable design in the sense that it is designed in anticipation of certain requirements that might arise in future. The

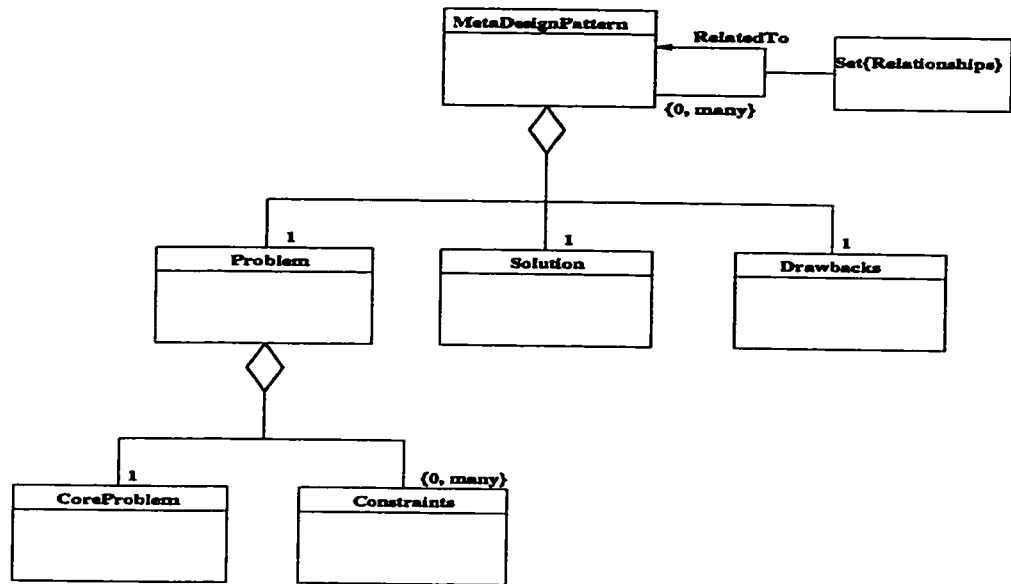


Figure 14: High Level Object Model For Database Schema

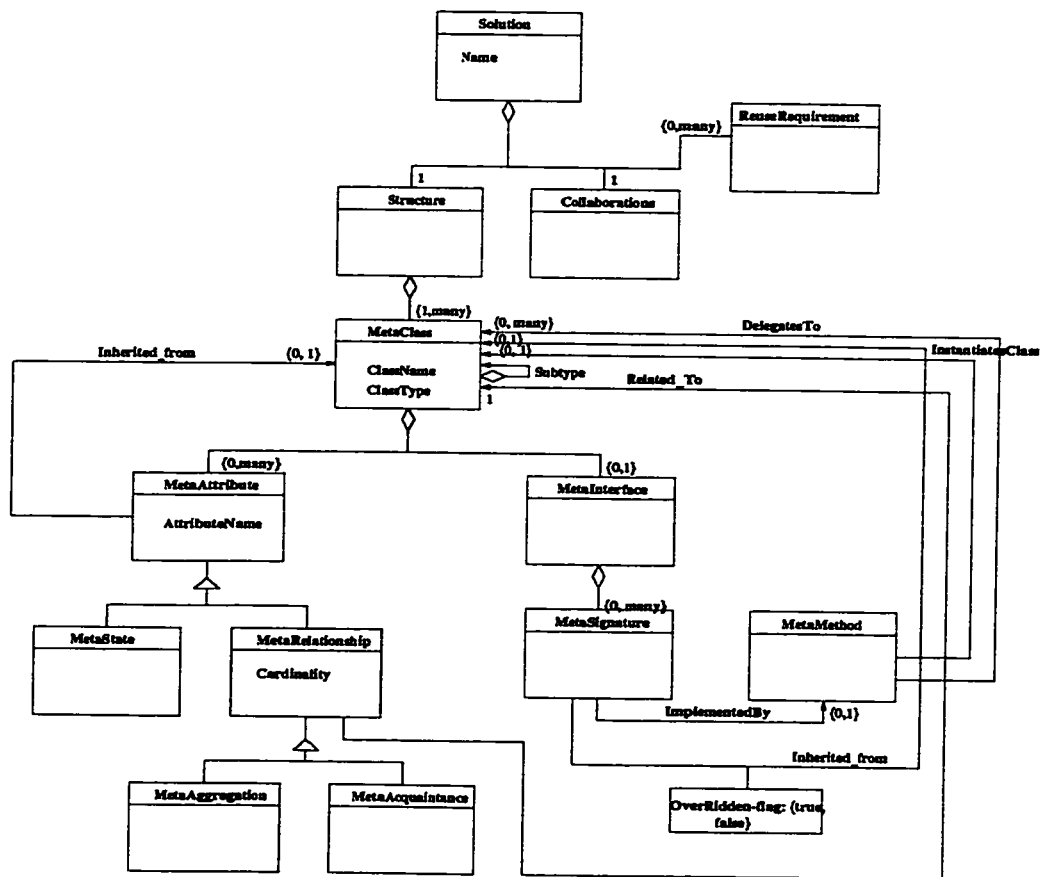


Figure 15: Object Schema for the Design Pattern Solution

design pattern solution is robust to handle such future requirements. We can therefore consider each instance of the `Solution` class as being an aggregate of a static structure modelled by the `Structure` class, dynamic aspects of the design pattern solution modelled by the `Collaborations` and future requirements that the design pattern solution can meet are modelled by the `Reuse Requirement`.

The static structure of design pattern solution represents the participant classes in the pattern, the relationship between these classes, description of their interfaces, inheritance hierarchies, information about which methods are overridden. It is clear that modelling the structure requires meta-modelling such concepts as a class, interface, instance variables, acquaintance and aggregation relationships, inheritance hierarchies. Our metamodel for the `Structure` represents a description of the intersection of concepts, and abstractions found in the set of design pattern structures found in [25]. Our meta-modelling efforts are motivated to a certain extent by the formalism that was discussed in Chapter 5. In meta-modelling these entities we shall confine ourselves to the design pattern solution' structures found in [25]. It is clear that the structure of design pattern solution can be thought of as a set of classes with certain relationships between them. We therefore model the `Structure` as an aggregation of one or more instances of `MetaClass` where each instance of `MetaClass` represents an abstract or concrete class. Each such instance of `MetaClass` has a name and a flag indicating whether the class is an abstract or concrete class. Besides each instance of `MetaClass` also represents a field that indicates the problem domain entity that it represents (for instance in the Observer design pattern, the Observer class implements the dependent objects). This information is useful to a designer in understanding the relationship between the abstract design pattern solution and the corresponding abstract problem statement. Each instance of the `MetaClass` is also an aggregate of its subtypes. This is indicated by the aggregation relationship `Subtype` shown in the Figure 15.

Each instance of the `MetaClass` consists of an interface and attributes. We therefore model `MetaClass` as an aggregate of classes `MetaInterface` and `MetaAttributes` that model abstractly the interface and attributes of an instance of `MetaClass`. Every attribute of a class in the design patterns found in [25] represents instance variables that characterize the state of objects in the class or represents object reference that implement the relationship of the class with other classes in the design pattern solution. For instance, the Memento pattern [25] denotes the instance variables which

characterize the state of the objects of the **Memento** class as **MementoState**. In the same pattern the **Originator** class is responsible for creating and knowing its **Memento**. This is thus captured in our model by the inheritance relationship between the **MetaAttribute** class and **MetaState** class and **MetaRelationship** class. The class **MetaRelationship** represents an acquaintance or aggregation relationship. Either of these relationships will have an associated cardinality. Though both relationships are usually implemented as pointers, there is a semantic difference. In particular, one-to-many acquaintance relationships are often confused with aggregation. However, the two are different. In the former case, objects within the relationship can have independent existence while in the latter an object in the relationship is responsible for other objects. For instance, the one-to-many relationship between a subject instance and an observer instance in the **Observer** design pattern is not an aggregation. The acquaintance and aggregation relationships are therefore modelled as separate classes and they are subclasses of the **MetaRelationship** class. Though relationships are owned by an instance of the **MetaClass** (ie. implemented as reference in a class), each such relationship relates the owner instance to one or more instances of **MetaClass**. This is represented in our schema by the **Related_To** relationship. Similarly each attribute in a class may be inherited from an instance of the **MetaClass** and this is indicated in the object model by means of the relation **Inherited_from**.

The **MetaInterface** class is modelled as an aggregate of signatures. This follows from the very definition of the interface as a set of signatures [25]. Also, many of the design patterns in [25] consist of abstract classes that only represent interfaces (abstract classes with pure virtual functions in C++). Each instance of the **MetaInterface** class represents a set of signatures. We model a signature as an instance of **MetaSignature** class. A class within a design pattern solution may or may not define an implementation for a signature. Further, the class can inherit the method from a superclass or override the method in the superclass. For this reason the methods are modelled as distinct entities that are instances of the **MetaMethod** class and each signature of a class can have at most one method implementing it. This is indicated by the **ImplementedBy** relationship in the Figure 15. A class can inherit the interface of a superclass, and it can either inherit the method associated with the signature or override it. This is indicated in the figure by the *Inherited_from* relationship between the **MetaSignature** class and **MetaClass**. Each instance of the **MetaSignature** class has

a name, certain arguments and a return type. Further, it may have an informal description associated with it. This is due to the fact that often in design pattern found in [25] the operations within a class have a highly abstract description and are not accompanied with a precise specification of their behavior – hence there is a need for this information to be associated with each signature.

The class `MetaMethod` models the methods implementing a signature in a given class. The instances of this class are represented in [25] as pseudocode. It is therefore not possible to have a formal model to characterize them. However, it would be useful to the reuser to query the database for design patterns that employ call delegation or class instantiation in a method. For instance, most of the Structural patterns in [25] use object composition and delegation. A reuser who is not aware of the particular category (in this case Structural) can still pose a query based on the object composition and delegation structure required. Typically a method in a class would delegate implementation to an instance variable, or to an argument of the signature with which the method is associated, or to local variables. In fact, the same information can be exploited to answer call delegation at the object level. Creational patterns in [25] usually employ methods that instantiate a class. We represent this information also in the `MetaMethod` class.

The class `Collaborations` models the collaboration among the participating classes in the design pattern as well as with a client object (an object that require the services of objects in the pattern). In Chapter 5 we provided a formal specification for collaboration aspects of a few design patterns in [25]. The formal specification provided us with a means for specifying collaborations in two dimensions: one dealt with collaboration occurring as a result of invocation of service from an object in the pattern (ie. an object that is an instance of a class from the design pattern), and another occurs when a client requests certain services from objects in the pattern (may be in a certain order) or configures objects in the pattern so that they can communicate. However, the collaborative aspects of design patterns are often too vague and in general cannot be represented as a formal model. For this reason we represent the `Collaborations` only by means of keywords and full-text and require the database system to provide querying facilities for a full-text search.

The class `ReuseRequirement` represents a reuse requirement. Each reuse requirement addresses a requirement that can arise in future. Each such requirement enables the designer to vary some aspect of the design without varying many other aspects

of the design. This essentially means that the changes to the design are minimal and local. The ReuseRequirement also provides the rationale for being able to vary an aspect without varying other aspects in the design. We therefore represent each ReuseRequirement as consisting of three components viz. a component that models the aspect that varies, a component that models the related aspects that do not vary and a component that provides the rationale. Each of these component can again be represented by means of key words and full-text.

6.6 Schema Definitions

In this section we provide more detailed representation for the entities described in the object models shown in Figures 14 and 15. We provide all the class definitions and each class definition is followed by a few method definitions. The list of method definitions provided for each class is not exhaustive.

class *MetaDesignPattern*

type tuple (Name: string, OtherNames: set(string), scope: enum{class, object}, purpose: enum{creational, structural, behavioral}, domains: set(tuple(domain:string, domain_contexts:Text)), frameworks: set(tuple(framework: string, framework_contexts: Text)), designproblem: Problem, designsolution: Solution, patterndrawbacks: Drawbacks, RelatedTo: set(tuple(pattern: MetaDesignPattern, underrelationship: Text)));

method *get_name* in class *MetaDesignPattern*: string

method *get_scope* in class *MetaDesignPattern*: scope

method *RelatedUnder* in class *MetaDesignPattern*:string

method *IsClassScope*() in class *MetaDesignPattern*:bool

method *IsCreational*() in class *MetaDesignPattern*:bool

method *IsFoundInDomain*(string) in class *MetaDesignPattern*:bool

method *GetContextInDomain*(string) in class *MetaDesignPattern*:bool

method *IsFoundInFramework*(string) in class *MetaDesignPattern*:bool

method *RelatedToBy*(MetaDesignPattern) in class *MetaDesignPattern*:string

class *Problem*

type tuple (core_problem: CoreProblem, constraints: set(Constraint));

class *CoreProblem*

type tuple (keywords: set[string], problem_text: text);

method foundkeyword(string) in class *CoreProblem*: bool

method getproblemtext in class *CoreProblem*: bool

class *Constraint*

type tuple (keywords: set[string], Designchoice: text, Rationale: text, constraint_text: text);

method getdesignchoice in class *Constraint*:string

method getrationalere in class *Constraint*:string

class *Drawback*

type drawback_text: text;

method getdrawbacks in class *Drawback*: string

class *Solution*

type tuple (static: Structure, dynamic: Collaborations, reusereqs: set(ReuseRequirement));

class *Structure*

type set(class:MetaClass);

method ContainsClass in class *Structure*: bool

method InheritsFrom(MetaClass, MetaClass) in class *Structure*: bool

method HasRecursiveAggregates in class *Structure*: bool

class *MetaClass*

type tuple(Name: string, ClassType: enum{Abstract, Concrete}, Represents: string, class_interface: MetaInterface (opt), class_attributes:set(MetaAttribute), Subtype: set(MetaClass));

method get_classtype in class *MetaClass*: ClassType

method get_responsibility in class *MetaClass*: string

method has_signature(string) in class *MetaClass*: bool

method has_subtype(MetaClass) in class *MetaClass*: bool

method has_acquaintance_with(string) in class *MetaClass*: bool

method DoesInstantiate(MetaClass) in class *MetaClass*: bool

```

class MetaAttribute
type tuple (owned_by: MetaClass, AttributeName: string, Inherited_from: MetaClass
(opt));
method get_name in class MetaAttribute: string
method IsInheritedFrom in class MetaAttribute: MetaClass

class MetaState inherit MetaAttribute

class MetaRelationship inherit MetaAttribute
type tuple(cardinality: enum {zero-one, zero-many, one-one, one-many}, Related_To:
MetaClass);
method hascardinality in class MetaRelationship: cardinality
method IsRelatedTo in class MetaRelationship: MetaClass

class MetaAggregation inherit MetaAttribute, MetaRelationship

class MetaAcquaintance inherit MetaAttribute, MetaRelationship

class MetaInterface
type tuple(owned_by: MetaClass, set(MetaSignature));
method has_signature(MetaSignature) in class MetaRelationship: bool

class MetaSignature
type tuple(Name: string, Arguments: List(MetaClass), ReturnType:MetaClass (opt),
Description:string, ImplementedBy: MetaMethod (opt),
InheritedFrom: tuple(superclass:MetaClass, over-riddenflag: enum{true, false}) (opt));
method getdescription in class MetaSignature : string
method isfoundindescription(string) in class MetaSignature : string

class MetaMethod
type tuple(Psudocode: string, DelgatesTo:set(tuple(Name:string,
TypeOfObject: enum{Reference, Argument, NotKnown}, OfClass:MetaClass)), In-
stantiatesClass: MetaClass (opt));
method DoesDelegateTo(string, MetaClass) in class MetaMethod: bool
method DoesInstantiate(MetaClass) in class MetaMethod: bool

```

```
class Collaborations
type tuple(keywords:set(string), collaboration_text:text);
method get_collabs in class Collaborations: string
```

```
class ReuseRequirement
type tuple(VaryingAspect:string, NonVaryingAspect:string, Rationale:text, reuse_requirement_text:text);
method CanVary(string) in class ReuseRequirement : bool
method get_rationale in class ReuseRequirement : bool
```

6.6.1 Integrity Constraints

Integrity constraints provide a means for ensuring that changes made to the database by authorized users do not result in a loss of data consistency. They thus guard against accidental damage to the database. For instance, cardinality constraints restrict the number of objects that are related to a given object. Domain constraints restrict the values of an attribute. Cardinality constraints are also examples of domain constraints. Implication constraints relate the values taken by attributes in a single object or related objects. Disjointness constraints express the constraint an instance of a subclass cannot move to another subclass in the same hierarchy. Here are a few examples of such integrity constraints found in our data model.

- **Cardinality Constraints:** These constraints are shown in the object models given in Figures 14 and 15. For instance, from Figure 15 we see that every design pattern solution's structure is composed of one or more instances of *MetaClass*.
- **Implication Constraints:** In the class *MetaMethod*, an implication constraint is found in the attribute *DelegatesTo*. This attribute has a *Name* field and *OfClass* field. The *OfClass* field refers to a class and the *Name* field refers to the name of a signature in this class to which an instance of *MetaMethod* is delegated. For example, if an instance of *MetaMethod*, has *DelegatesTo* attribute : (*foo*, *Reference*, *ArbitraryClass*), then *foo* must be a signature in the interface of *ArbitraryClass*. Similarly, the *RelatedTo* attribute of *MetaDesignPattern* has an integrity constraint: if the *RelatedTo* attribute of design pattern A contains a tuple : (pattern B, *Relationship_AB*) (ie. A is related to B under a relationship

AB) then the *RelatedTo* attribute of design pattern B also has the tuple : (pattern A, Relationship_AB) (ie. B is related to A under the reverse relationship). For instance, the Abstract Factory pattern is related to the Factory Method pattern since the latter is often used to implement the former. This relationship appears in the *Related* attributes of both patterns.

- **Disjoint Constraints:** Object models capture some constraints through their very structure. For instance, the single inheritance hierarchy for *MetaRelationship* in the object model 15 implies that the subclasses *MetaAggregation* and *MetaAcquaintance* are mutually exclusive.

6.6.2 An Example

The Observer design pattern [25] is one of the most popular and non-trivial design patterns. As has been explained in earlier chapters, the pattern is found in many GUI applications for decoupling the model from the presentation. In this section we consider the Observer design pattern as an example to illustrate the instantiation of the OO schema.

In our schema, the Observer pattern is an instance of *MetaDesignPattern* class. Its scope and purpose are *Object* and *Behavioral* respectively. It is usually found in GUI domains in the context of decoupling the representation of the model from the presentation. It is also found in the MVC framework [36] where Model class plays the role of the Subject and View is the base class for observers, Unidraw framework where graphical objects are split into View (for observers) and Subject parts [25]. The Observer pattern is related to *Mediator* and *Singleton* patterns and the nature of this relationship is specified in [25]. These details are described by the attributes of the Observer instance (which is an instance of *MetaDesignPattern*). The *CoreProblem* can be represented by means of a set of keywords such as *dependency*, *one-to-many* which capture the essence of the problem. The full-text of the problem can also be represented. The Observer pattern has a single design constraint specified in Chapter 5 and this states that the design pattern solution must avoid tightly coupling the objects. The rationale for the constraint is that tightly coupling the objects would prevent them from being reused independently. These details are represented by attributes of an instance of *Constraint* class.

The *Structure* part of the *Solution* instance has four instances of the *MetaClass*.

These are Subject, ConcreteSubject, Observer and ConcreteObserver. Subject and Observer are abstract classes, while ConcreteSubject and ConcreteObserver are concrete classes. The Subject class represents the interface of subject. An informal description of its responsibilities is provided by the attribute Represents. It has only one attribute, viz. an one-many acquaintance relationship with Observer Class and this is represented by an instance of MetaAcquaintance with attributes: owned_by: Subject, AttributeName:observers, cardinality:zero-many, Related_To: Observer. Its interface provides the signatures Attach(Observer), Detach(Observer) and Notify which are instances of MetaSignature class. A pseudocode is provided only for Notify and this is represented in Notify instance by the attribute:ImplementedBy. This pseudocode will be an instance of MetaMethod class and its attributes are: Pseudocode which represents the pseudocode, DelegatesTo: {tuple(Update(), ObjectReference, Observer)}. Similarly, ConcreteSubject, Observer and ConcreteObserver are represented. The collaborations in the Observer pattern is an instance of Collaborations class and this instance has attributes representing the collaborations in terms of key-words and free-text.

The Observer pattern has two reuse requirements as explained in Chapter 5. One of these is that the type of observer objects can be varied independent of the subject. The rationale for this is that, the subject and observers are abstractly coupled, hence new types of observers can be introduced by subclassing Observer class and without affecting the Subject. For the same reason, the type of observer objects dependent on a subject can be varied at run-time. This reuse requirement is thus an instance of Reuse Requirement and has attributes: VaryingAspect: Type of observer, Non-VaryingAspect: Subject. This instance also has a Rationale attribute that represents the rationale (provided above) for the reuse requirement as free-text.

Chapter 7

Conclusions

In this thesis we have emphasized the need for a formal approach to documenting classes, micro-architectures and design patterns which arise in the OO framework for developing software systems. After reviewing the levels of reuse and existing documentation styles for design patterns, we introduced formal specification styles for C++ classes, micro-architectures, design patterns and a schema for a design pattern repository.

The thesis has made the following contributions for reusing software designed using OO principles:

- Building on the seminal work of Colagrosso [16], Larch/C++ language is extended with constructs for expressing object-collaborations. This extended three-tiered language is used to specify micro-architectures.
- A critique on existing design pattern documentations, a rationale for a new documentation, and a formalism for essential features of design patterns has been provided.
- Software reuse in large scale is feasible when software artifacts are stored and retrieved from a database. The thesis presents an object-oriented database schema, and possible queries that can be posed on the database of design patterns.

The design of our design pattern repository has addressed the following key issues:

- **Classification:** Design patterns are classified across more than one dimension. Besides the classification provided by Gamma et al, the patterns are also classified under the domains in which they occur and the frameworks that use them. Besides, retrieval of patterns related under different criterias is also possible.
- **Storage:** Each design pattern is stored as an object in a persistent object-oriented database. The components of such a design pattern are all stored as objects, thereby providing a finer representation of such components and making use of object-oriented concepts such as encapsulation and inheritance.
- **Retrieval:** The informal documentation of pattern is provided in a high-level formal structure in our schema. Further, the information is organized in a logical and intuitive fashion. It is possible to provide a query language that allows users to perform queries based on design pattern attributes, as well as traditional string and keyword matching. The query language should enable users to pose conjunctive and disjunctive queries as described in sections 6.2.1 and 6.2.2. We believe that our schema is rich enough to handle all the queries outlined in sections 6.2.1 and 6.2.2. Besides, query languages can be designed to query information using methods attached to the classes in the schema.

The following areas of future work arise from this thesis:

- (1) The notion of class behavior and completeness evaluation of C++ classes was addressed in [16]. If micro-architecture is to be treated as a unit, then the specifier should have a complete understanding of the behavior of the micro-architecture. To ensure this, one has to define the notion of behavior of a micro-architecture, define completeness and provide an evaluation methodology for evaluating the completeness of micro-architecture specifications.
- (2) A methodology for conformance testing of C++ classes with respect to their Larch/C++ specifications was provided in [11]. This methodology needs to be extended to perform conformance test on micro-architecture implementations against their specifications.

- (3) More research is required in evaluating the adequacy of the formal methodology given for design patterns.
- (4) We believe that the conceptual schema provided for the design pattern repository can be implemented as is in an object-oriented database management system (OODBMS), suitably modifying it to suit a specific OODBMS and considering query optimization and efficiency issues in such a system.
- (5) A major interesting work would be the design of a query language with deductive capabilities for retrieving design patterns from the database designed according to our schema.
- (6) In this thesis we have confined ourselves to the conceptual aspects of the database design. It would be interesting to go further and look at the logical and physical design of such a repository. Further, it would be possible to use the design patterns in [25] in the *physical* design. The recursive nested structure of data points to feasibility of using the Composite pattern. Similarly the navigation type of queries may require the use of Iterator pattern.
- (7) In the description of the OO schema, we gave some examples of semantic integrity constraints associated with design patterns without indicating how they can be exploited for data and knowledge retrieval. A possible extension to this research consists of studying the way integrity constraints can be used as a support for intelligent retrieval of relevant data, and as background knowledge to discover clusters and relationships between design patterns using documentation components as input. Such mining techniques can help the designer discover links between design patterns and retrieve more easily the ones that are most relevant to his needs.
- (8) Designing a web interface as a front-end to such a repository would make the retrieval of information from design patterns all the more interesting and attractive.
- (9) Since design pattern documentation includes attributes in a full-text form, appropriate techniques for indexing and retrieving information from such data types can be exploited using well-spread tools such as those provided by some database management systems (e.g. Oracle).

Bibliography

- [1] V.S. Alagar, P.Cologrosso, A. Loukas, S. Narayanan, A. Protopsaltou. *Formal Specifications for Effective Black-Box Reuse-Revised Phase I Report*. Report submitted to Nortel, Nuns Island, Montreal, Canada, March 1996.
- [2] V.S. Alagar, P.Cologrosso, A. Loukas, S. Narayanan, A. Protopsaltou. *Formal Specifications for Effective Black-Box Reuse-Phase II Report*. Report submitted to Nortel, Nuns Island, Montreal, Canada, March 1996.
- [3] P.S.C Alencar, D.D. Cowan, D.M. German, K. J. Lichtner, C.J.P Lucena, L.C.M. Nova. *A formal approach to design pattern definition and application*. ECOOP 96.
- [4] P. America *Inheritance and subtyping in a parellel object-oriented language*. In Jean Bezevin, Jean-Marie Hullot, Pierre Cointe, and Henry Lieberman, editors, ECOOP'87, Paris, France, 1987. Springer-Verlag. Lecture notes in Computer Science 276.
- [5] W. Bartussek and D.L. Parnas *Using assertions about traces to write abstract specifications for software modules*. In Gehani and McGettrick, Editors, Software Specification Techniques, Addison Wesley, 1986.
- [6] G. Booch *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1994. Second Edition.
- [7] M. Bidoit. *A generalization of initial and loose semantics*. Technical Report 402, Orsay, France, 1988.
- [8] C.L. Braun. *Reuse in Encyclopedia of software engineering*. J. Marciniak (ed), John Wiley and Sons, 1066-1069, 1994.

- [9] L. Cardelli and P. Wegner. *On understanding types, data abstractions and polymorphism*. ACM Computing Surveys, 17(4), December 1985.
- [10] R.G.G. Catell, T.R. Rogers. *Combining object-oriented and relational models of data*. International Workshop on Object-Oriented Database Systems, Pacific Grove, Calif., Sept. 1986.
- [11] A. Celer. "Role of formal specifications in black-box testing of object-oriented software". Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1995.
- [12] M.Cline and D. Lea. *The behavior of C++ classes*. In Proceedings of Object-Oriented Programming Emphasizing Practical Applications. Marist College, 1990.
- [13] M.Cline and D. Lea. *Using annotated C++*. In Proceedings of the 1990 C++ At Work Conference, 1990.
- [14] P. Coad. *Object-Oriented Patterns*. Communications of the ACM. September 1992, Vol. 35, No. 9.
- [15] P. Chalin. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language*. PhD Thesis, Department of Computer Science, Concordia University, 1996.
- [16] P. Cologrosso. *Formal specification of C++ class interfaces for software reuse*. MSc Thesis, Department of Computer Science, Concordia University, 1993.
- [17] B. Cox. *Message-object programming: An evolutionary technology*. IEEE Software, January 1984.
- [18] B. Cox. *Object-oriented programming: An evolutionary approach*. Addison-Wesley, Reading, Ma., first edition, 1986.
- [19] B. Cox. *Planning the Software Industrial Revolution*. IEEE Software, November 1990.
- [20] S. Danforth and C. Tomlinson. *Type theories and object-oriented programming*. ACM Computing Surveys, 20(1), March 1988.

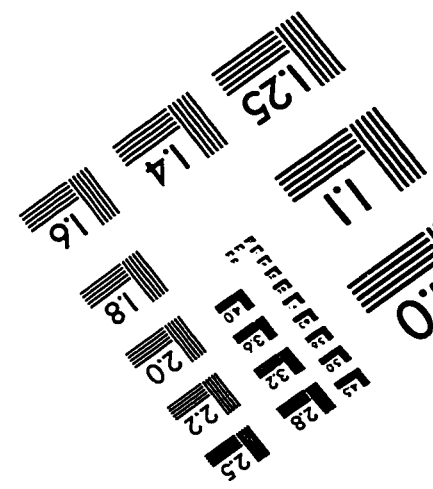
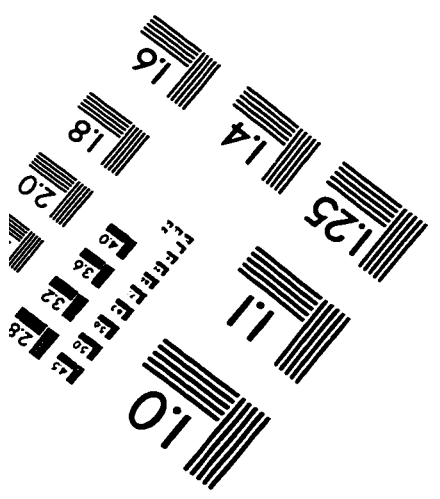
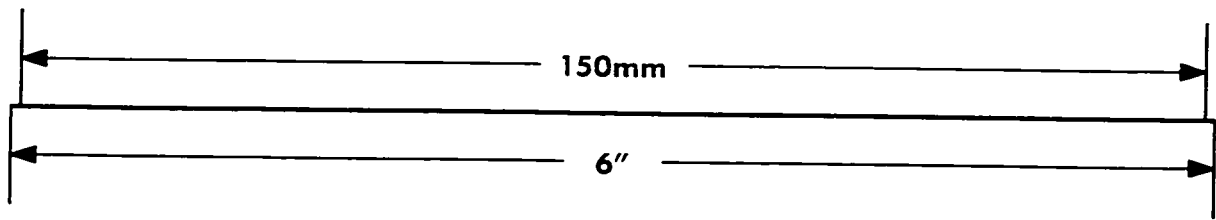
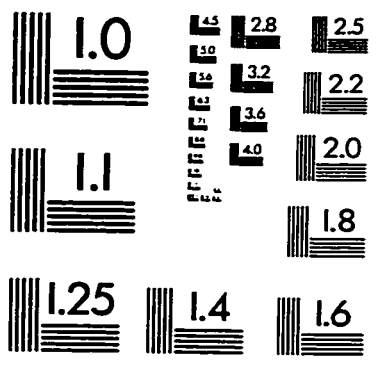
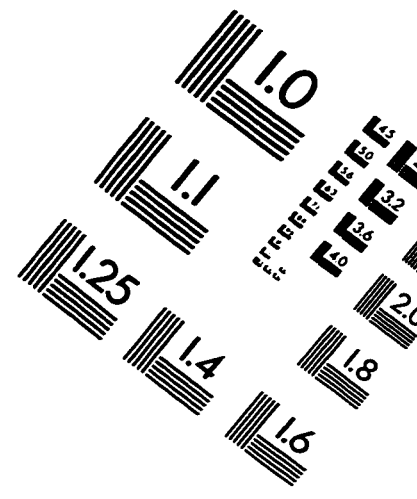
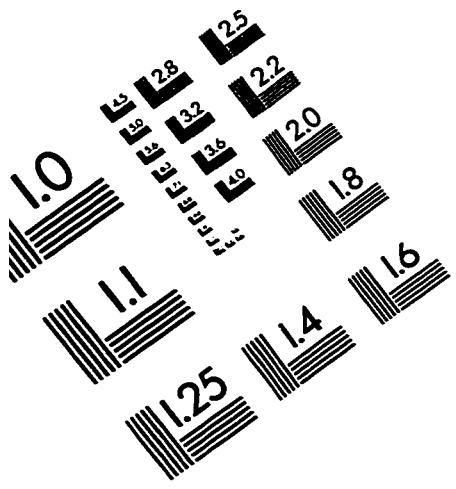
- [21] D. de Champeaux, P. America, D. Coleman, R. Duke, D. Lea, and G. Leavens. *Formal techniques for OO Software Development (panel)*. In proceedings of OOPSLA' 1991, 1991.
- [22] H. Ehrig, and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer Verlag, 1985. EATCS Monographs on Theoretical Computer Science, vol. 6.
- [23] W. Frakes, and S. Isoda. *Success factors for Systematic Reuse*. IEEE Software, 15-19 September 1994.
- [24] J.E. Gaffney and R.D. Gruickshank. *A General Economics Model of Software Reuse*. Proceedings of the 14th International Conference on Software Engineering, IEEE Computer Society Press, Los Altimos, CA, 327-337 1992.
- [25] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Inc. 1995.
- [26] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. ECOOP'93 Conference Proceedings, Springer Verlag Lecture Notes in Computer Science.
- [27] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*. Prentice Hall, 1978.
- [28] A. Goldberg and D. Robson. *Smalltalk-80, The Language and its implementation*. Addison Wesley Publishing Co, Reading, Mass. 1983.
- [29] J.V. Guttag and J.J Horning. *Larch: Languages and Tools for Formal Specification*. Springer Verlag 1993.
- [30] R. Helm, Ian M. Holland and D. Gangopadhyay. *Contracts: Specifying behavioral compositions in object-oriented systems*. ECOOP/OOPSLA ' 90 proceedings, October 21-25, 1990.
- [31] D. Ince and D. Andrews. *The Software Life Cycle*. Open University Press, 1990.

- [32] R. Joos. *Software Reuse at Motorola*. IEEE Software, 42-47, September 1994.
- [33] Ralph E. Johnson and Brian Foote. *Designing Reusable Classes*. Journal of Object-Oriented Programming, 4(2):22-35, June/July 1988.
- [34] Ralph. E. Johnson. *Documenting Frameworks using Patterns*. OOPSLA' 92, pp. 63-76.
- [35] T. Korson and J.D. McGregor. *Understanding Object-Oriented : A unifying paradigm*. Communications of the ACM, 33(9), September 1990.
- [36] Glenn E. Krasner and Stephan T. Pope. *A cookbook for using the model-view controller user interface paradigm in SmallTalk-80*. Journal of Object-Oriented Programming, 1(3):26-49, August/September 1988.
- [37] R. Lajoie, R.K. Keller. *Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts, and Motifs in Concert*. Object-Oriented Technology for Database and Software Systems, Eds. V.S. Alagar, R. Missaoui. World Scientific Publishing Co. Pte. Ltd, 1995.
- [38] G. Leavens, Y. Cheon. *A Quick Overview of Larch/C++*. TR No. 93-18, Department of Computer Science, Iowa State University, June (1993).
- [39] G.T. Leavens. *Modular verification of object-oriented programs with subtypes*. Technical Report 90-09, Department of Computer Science, Iowa State University, July 1990.
- [40] G.T. Leavens. *Modular specification and verification of object-oriented programs*. IEEE Software, 8(4), July 1991.
- [41] G.T. Leavens and W. E. Weihl. *Reasoning about object-oriented programs that use subtypes*. SIGPLAN notices, 25(10), October 1990. Proceedings of ECOOP/OOPSLA '90.
- [42] W.C. Lim. *Effects of Reuse on Quality, Productivity and Economics*. IEEE Software, 11, 23-30 1994.
- [43] Mark. A. Linton, J. Vlissides, Paul. R. Calder. *Composing user interfaces with InterViews*. Computer, 22(2)8-22, February 1989.

- [44] B. Liskov. *Data abstraction and hierarchy*. ACM SIGPLAN Notices, 23(5), May 1988. Revised version of the keynote address given to OOPSLA'87.
- [45] D. Luckham. *Programming With Specifications: An Introduction To Anna, A Language for Specifying ADA Programs*. Springer-Verlag 1991.
- [46] David Maier, Jacob Stein, Allen Otis, Alan Purdy. *Development of an object-oriented DBMS*. OOPSLA' 86 as ACM SIGPLAN 21, 11, Nov. 1986, 472-482.
- [47] M.D. McIlroy. *Mass Produced Software Components*. Software Engineering: Report on a Conference by Nato Science Committee (Garmish, German), P. Naur, D. Randell, EAS, NATO Scientific Affairs Division, 138-150, Brussels, Belgium 1968.
- [48] B. Meyer. *Object-Oriented Software Constructions*, Prentice Hall, 1988. Prentice-Hall, 1988.
- [49] Robert T. Monroe and David Garlan. *Style-Based Reuse for Software Architectures*. Fourth International conference on Software Reuse, Orlando, Florida, April 23-26, 1996.
- [50] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley Publishing Company, ACM press, 1995.
- [51] *Rogue Wave, 'Tools.h++ Class Library v6.0'*. Rogue Wave Software (1993).
- [52] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.
- [53] D.T. Sanella and A. Tarlecki. *On observational equivalence and algebraic specification*. *Journal of Computer and System Sciences*, 34:150-178, 1987.
- [54] D. Schmidt. *Introduction to Design Patterns*. Tutorial presented at: Short-course on Design Patterns at the Open Computing Institute, St. Louis, MO.
- [55] D. Schmidt. *Experience using design patterns to develop reusable object-oriented communication software*. Communications of the ACM, Special issue on Object-Oriented experiences. Vol. 38, No.10, October 1995.

- [56] D. Schmidt and James O. Coplien. *Pattern languages of program design*. Reading, Mass. : Addison-Wesley, 1995.
- [57] I. Sommerville. *Software Engineering*. Addison Wesley, third edition, 1989.
- [58] A. Snyder. *Encapsulation and Inheritance in object-oriented languages*. OOPSLA, pages 38-45, Portland, OR, November 1986. ACM press.
- [59] T.A. Standish. *An essay on software reuse*. IEEE Transactions on Software Engineering, 10(5), September 1984.
- [60] A. Weinand, E. Gamma, and R. Marty. *ET++ - An object-oriented application framework in C++*. In OOPSLA, pages 46-57, San Diego, CA, September 1988.
- [61] C.P. Willis. *Analysis of inheritance and multiple inheritance*. IEEE 1996.
- [62] A. Wills. *Specification in Fresco*. In S.Stepney, editor, *Comparitive Study of Object-Oriented Specification Methods*. Prentice Hall, 1992.
- [63] J. Wing. *A specifier's introduction to formal methods*. IEEE Computer, 23(9), September 1990.
- [64] Rebecca J. Wirfs-Brock and Ralph E. Johnson. *Surveying current research in object-oriented design*. Communications of the ACM. September 1990, Vol.33, No.9.
- [65] Dr. Won Kim. *A UniSQL Whitepaper*.
- [66] C. J. Woodcock and R.G. Clark. *Defining Object-Oriented Design Patterns Within Frameworks*. Technical Report CS-130, Department of Computing Science and Mathematics. University of Sterling, November 1994.
- [67] C. Zaniolo, H. Ait-Kaci, D. Beech, S. Cammarata, L.Kerchberg, and D. Maier. *Object-oriented database and knowledge systems*.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved