FORMAL SPECIFICATION OF DESIGN PATTERNS:

A COMPARISON OF THREE EXISTING APPROACHES AND

PROPOSING TWO-LEVEL GRAMMARS AS A NEW APPROACH


By

DEEPA BALASUNDARAM

Bachelor of Technology in Information Technology

Bharathidasan University

Tiruchirapalli, Tamilnadu

2006


Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 2010

FORMAL SPECIFICATION OF DESIGN PATTERNS:

A COMPARISON OF THREE EXISTING APPROACHES AND

PROPOSING TWO-LEVEL GRAMMARS AS A NEW APPROACH


Thesis Approved:

Dr. M. H. Samadzadeh
_____
Thesis Advisor
Dr. John P. Chandler
_____


Dr. Blayne Mayfield
_____


Dr. A. Mark E. Payton
_____
Dean of the Graduate College

PREFACE


Patterns are Object-Oriented reusable units. The principal idea behind patterns is to capture and reuse the abstractions that have been formed by expert programmers and designers to solve problems that occur in particular contexts. These abstractions capture the valuable experiences of experts in solving problems. Although patterns are currently being used successfully, there is no general agreement among the software community as to how patterns should be formalized or represented. Various formal specification schemes have been proposed to complement the natural language description of patterns in order to alleviate the ambiguities inherent in the natural language description by rigorously reasoning about the structural and behavioral aspects of patterns. Existing formal specification languages of design patterns have generally failed to provide a standard definition, specification, or representation for patterns because there is no general agreement as to how patterns should be formalized. Also, each formal specification is generally based on a different mathematical formalism and when pattern users want to understand a pattern, first they have to understand the respective mathematical formalism.

In addition to comparing three existing formal specification schemes, the main objective of this research work was to lay the foundation for developing a formal specification scheme that could be understandable without having to delve into the details

of the underlying formalism. This research work attempted to capture and represent the structural aspects of design patterns since capturing the behavioral aspects of design patterns is a semantic issue and is beyond the scope of this work. Two-Level Grammar (TLG) was used to capture and represent the structural aspects of design patterns. This study was conducted using the GoF design patterns [Gamma et al. 1995]. It has already been demonstrated that TLGs have the capability to represent the building blocks of object-oriented software systems. The primary advantage of TLGs in defining design patterns is that specifications written in TLGs are understandable due to their natural-language-like vocabulary [Edupuganty 1987] [Lee 2003] [Maluszynski 1984]. The TLG representation of the observer pattern was developed to gauge the feasibility of the proposed pattern representation scheme. TLGs could help pattern users understand the formalized version of patterns more readily compared to other formal specification methods that are difficult to understand due to their arcane mathematical notations.

# ACKNOWLEDGMENTS

A famous Sanskrit phrase delineates the order of priorities for a child, "Matha, Pitha, Guru, Deivam (mother, father, teacher, and god)", implying that first it is mother, then father, then teacher, and, only after all of them, God. In a sense, mother, father, and, guru show the children the path to God. I am very fortunate to have the best mother, father, and guru one could ever have in their life.

It is believed in Hinduism that self-realization cannot be achieved without the guidance and blessings of a proper guru. My advisor, Dr. Samadzadeh has been a proper guru. Thus, I would first like to express my gratitude towards Dr. Samadzadeh for showing enormous trust in me. Without his guidance, support, and constant encouragement, this research work would have been impossible and my studies in the US would have been quite difficult. His unique teaching skills and guidance towards the thesis and constant encouragement has been the driving force of my learning experience.

My sincere thanks to my committee members, Dr. Chandler and Dr. Mayfield, for providing valuable input during the proposal meeting. I should thank Dr. Amon Eden and Dr. Ralph Johnson who have always clarified my doubts related to design patterns whenever I contacted them.

I am deeply indebted to my parents and sister for their love, support, prayers, and constant encouragement throughout my life. My Success in each and

every step of life would have been impossible without them. Very special thanks goes to my fiancée, Karthic, who has been the source of inspiration for me and who has been very patient and understanding, and who was with me whenever I needed any kind of help. I am very thankful to my friends Malar and Neena who were the actual sources of inspiration for my pursuing graduate studies in the US. I would also like to extend my thanks to my friends Anusha, Bhavna, Janet, and Sudha for their help and support.

I would also like to express my thanks to few of my friends in the department, Alireza, Arif, Jaro, Mun, Peyman, Senthil, and Richard who have helped me in this thesis by asking probing questions. Without these friends and Friday group meetings, this piece of work would have been less enjoyable.

Finally, my sincere thanks go to Mr. Gary Kearans, Dr. Joyce Lucca, the Computer Science Department, ITLE, and CREC for partially funding my graduate studies. Without their financial support, my stay in US would have been virtually impossible.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

1.1 Introduction

Increased demand for software application development has led programmers to explore all possible reuse techniques. The potential benefits of software reuse include reduced development time and costs, shortened time-to-market, and improved software quality and maintainability [Schmid and Verlage 2002]. As a result, reusable software artifacts are being widely used by the software industry to build software systems faster to satisfy the ever increasing expectations from the user community. The notion of patterns can be considered as one of the possible software reuse artifacts.

Patterns are problem solving approaches based on urban planning and architecture [Alexander et al. 1977]. The principal idea behind patterns is to capture and reuse abstractions formed by expert programmers and designers to solve problems that recur in particular contexts and communicate the design knowledge in a domain-independent way. Alexander found recurring themes in building architectures that he captured into abstractions called "patterns". This pattern concept was adopted by the software community because of its straightforward relationships with the object-oriented constructs.

Software patterns are usually described as common solutions to recurring

software design problems [Gamma et al. 1995]. Patterns are gaining increasing acceptance and usage because they are abstractions generated from the valuable experiences of developers in solving problems that are repeatedly encountered in certain contexts [Buschmann et al. 2006]. These abstractions capture the valuable experiences of experts in solving problems. Since Patterns have generally been extensively tested and used in many development efforts, reusing them should yield better quality software within a reduced time frame [Taibi and Ling 2003 B]. They also capture the overall design experience in such a way that they have become a learning aid for novice designers [Taibi 2006].

1.2 The Problem

In the early stages of pattern evolution, patterns were described only by using *pattern forms* [Gamma et al. 1995]. Pattern forms define the essential elements of patterns using textual descriptions, sample code fragments, and graphical modeling languages. These descriptions are mostly in a natural language which is inherently informal, ambiguous, and sometimes misleading when used in an attempt to understand them. This is mainly due to the inaccurate and mostly vague verbal specifications, which cannot be definitive. As a result, pattern users were forced to understand the meaning of patterns from the interpretation of their verbal specifications [Eden 2000]. Hence, there was a need to formalize patterns in order to describe them accurately, reason about them rigorously, and also to facilitate tool support [Agerbo and Cornils 1998] [Bayley and Zhu 2008] [Buschmann et al. 2007] [Chinnasamy 2000] [Dong 2002] [Eden 2000] [France et

al. 2004] [Kent and Lauder 2004] [Mikkonen 1998] [Soundarajan and Hallstrom 2004] [Taibi and Chek Ling 2003].

Various formalization schemes have been proposed to describe design patterns accurately, in order to reason about them rigorously and to lead forward tool support for them. These formal specification schemes were expected to lay a foundation for tool support by clarifying the notions underlying patterns through rigorously reasoning about the structural and behavioral aspects of patterns. However, the existing formalization methods have failed to capture the essential structural and behavioral elements of patterns, which has led to a situation where there is neither a standard methodology for representing patterns nor a standard definition for *what a pattern is* [Agerbo and Cornils 1998] [Bayley and Zhu 2008] [Buschmann et al. 2007] [Chinnasamy 2000] [Dong 2002] [Eden 2000] [France et al. 2004] [Kent and Lauder 2004] [Mikkonen 1998] [Soundarajan and Hallstrom 2004] [Taibi and Ling 2003 B]. This is mainly due to the reason, as mentioned by Taibi and Ling [Taibi and Ling 2003 B], that, more often than not, each specification scheme is based on a different mathematical formalism that reflect each specific author's opinion on *how patterns should be formalized.*

A consequence of this deficiency in capturing the essential structural and behavioral elements of patterns may be that the number of patterns will increase to a level that it will become impossible to maintain the information as to which pattern solves which problem. This can also negatively impact the possibility of using patterns as a common vocabulary, and furthermore it will likely become hard to find the appropriate pattern(s) for a given problem. Moreover, the lack of a standard definition for patterns has led to a situation where concepts such as schemas, interfaces, client/server

communication, interaction, and composition are being viewed under the general rubric of patterns. It is necessary to determine the properties of patterns that would help in distinguishing patterns from other reusable units and also in restricting the formation of overlapping patterns.

Fundamentally, the problem resides in expressing the meaning of design patterns in a definitive way. Rigorous reasoning about the structural and behavioral semantic elements of design patterns is required to capture the essential properties of patterns, to represent them unambiguously, and to provide tool support, all of which can only be achieved by providing a formal model [Chinnasamy 2000] [Eden et al. 2007] [Mikkonen 1998] [Taibi and Ling 2003 B] [Taibi 2007]. Capturing the essential properties of patterns could address the problem of formation and proliferation of overlapping patterns, which could help in preserving the benefits offered by design patterns. The contention is that the traditional approaches used to define the semantics of programming languages can be used to capture the semantics or the structural and behavioral elements of design patterns accurately. The captured semantics or elements should be represented in such a way that it would be understandable (by everyone including the software developers who may not want to delve into the details of the mathematical foundation of a formalism before they can understand the meaning of a pattern) and less ambiguous. It is also important that the specification preserve the flexible nature of design patterns because when the specification tightens the base of a pattern, then it is not a pattern anymore.

1.3 The Approach

The work reported in this thesis formalizes the structural aspects of design patterns by keeping in mind the following critical attributes of a specification language: flexible nature of the patterns should be preserved, the specification scheme should be understandable and not complex to the users of patterns, and the mathematical concepts used for the formal specification schemes should not be a burden for the pattern users so that the pattern users will not have to unnecessarily delve into the as potentially prohibitive details of a mathematical formalism before they can understand the meaning of a pattern.

By formalizing the structural aspects of design patterns, this thesis work lays the foundation for a formal model for patterns that will become a complete model when the behavioral aspects of design patterns are captured and represented, along with tool support and pattern repository management schemes. The contention is that when the structural and behavioral semantics of patterns are captured, it should be straightforward to capture the essential elements of design patterns accurately, thus controlling and restricting the formation of overlapping patterns.

So, it is crucial to select a formalism to capture and specify the behavioral and structural elements of patterns accurately. This research work selected two-level grammars. Each design pattern is formed using a set of symbols. This set of symbols can be viewed as a language associated with that pattern. Grammars and other semantic formalization approaches have been used to formalize and generate programming languages without restricting the usability of these languages. Admittedly, programmers in general may not be directly and consciously aware of the formal grammars

underpinning the programming language that they use. Nonetheless, when one learns a programming language, it is typically through syntax charts and specific language construct templates which are in effect based on the formal grammars used to define programming languages and to parse the resulting programs. Therefore it seems appropriate to provide a formal specification of patterns using grammars and traditional semantic definition approaches that are used for languages.

The proposed formal specification scheme will only replace the ambiguous textual description of the design patterns, and it will not restrict the usability of patterns. In this thesis, two-level grammars (TLGs) were used to represent the syntactic or structural elements of design patterns. The capability for data and procedural abstractions, provided by the different levels of TLGs, makes TLG suitable for representing each level of refinement used in the concrete realization of patterns. TLGs can be used to represent different level of abstraction represented by design patterns by utilizing the concept of inheritance in object-oriented programming. For example, a TLG can be used to represent an abstract design pattern and, when additional levels of detail need to be included, those details can be added through inheritance or interface implementation.

The primary advantage of using TLGs to represent design patterns is that specifications written in TLGs are understandable due to their natural-language-like vocabulary [Edupuganty 1987] [Bryant and Pan 1992]. Thus TLGs could help pattern users understand the formalized version of patterns more readily compared to the other formal specification methods, such as BPSL and DisCo, that are difficult to understand due to their generally arcane mathematical notations. The close correspondence between

6

the natural language description of patterns and the TLG specification of patterns could make the TLG specification of design patterns widely acceptable. Since TLGs have already been used to formalize programming languages, it could be argued that the TLG representation of design patterns will not restrict the flexibility of design patterns [Edupuganty 1987].

1.4 Contributions

In addition to comparing three existing formal specification schemes, the work presented in this thesis report provides a basis for the formal specification of design patterns using two-level grammars. More specifically, this thesis presents an approach to formalize the structural aspects of design patterns using two-level grammars. The investment of time and effort, and the issue of complexity involved in the process of understanding and implementing design patterns can be reduced due to the natural-language-like nature of two-level grammars. Also, the ambiguous, and the unreliable nature of current pattern descriptions can be addressed, complemented, and made more understandable with the proposed formal specification scheme.

This work also lays a foundation for capturing the behavioral aspects of patterns. More specifically, by formalizing the structural aspects, this thesis lays the foundation for a formal framework that could be a complete model when the behavioral aspects of the design patterns are also captured and represented, along with the requisite tool support and repository management scheme.

1.5 Organization of the Thesis

The organization of the rest of this thesis report is as follows. Chapter II provides an overview of design patterns, including a brief discussion of the history of object-oriented design patterns. The GoF representation [Gamma et al. 1995] of the observer design pattern is included along with an overview of the existing formal specification schemes, namely, DisCo, BPSL, and LePUS3 and Class-Z with their representation of the observer pattern, in order to illustrate the ambiguities inherent in the natural language representation of design patterns and the need for formal specification schemes. This chapter also provides a brief description of the Two-Level Grammars (TLGs) and some of the existing applications based on TLGs. Chapter III introduces TLGs in the context of classes and objects. Chapter IV illustrates the use of TLG as a formal specification language to represent patterns by using the observer design pattern as an example. Chapter V summarizes the contributions of this research work and outlines some directions for future work.

# CHAPTER II

# LITERATURE REVIEW

2.1 History of Patterns

Patterns, as a concept, originated from the work of Christopher Alexander. Christopher Alexander found recurring themes in (building) architectures and captured them into descriptions that he called *patterns* [Alexander et al. 1977]. Christopher Alexander defined a pattern as a rule that describes "a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander et al. 1977]. The intention was to teach the language of the architects to everyone, so that even non-architects get the power and wisdom to bring liveliness to the places they live by designing their buildings and communities in the way they want. A pattern language, like a natural language, allows its users to create "an infinite variety of unique combinations [of its elements], appropriate to different circumstances, at will" [Alexander et al. 1977]. The elements of the pattern language are nothing but patterns. A pattern language captures the collective wisdom of the architects in terms of patterns [Alexander et al. 1977].

A pattern language is not merely a catalogue of patterns, it is a body of patterns that follows a strategic way to lead from a small pattern to larger ones providing the flow.

that connects various patterns. So, when a person is faced with a need to design, she/he does not have to start from scratch, rather they can learn or grasp the basic ideas from the experience of successful architects in terms of patterns, by learning the pattern language Therefore, a pattern language was used as a tool to share and communicate design knowledge in a domain independent way.

Patterns were described in a natural language, along with pictures of instances of the patterns, since patterns were meant to be a tool to communicate design knowledge in a domain independent way. Patterns abstracted by Alexander have the same format [Alexander et al. 1977]. First there is a picture that shows the instance of a particular pattern, followed by an introductory paragraph that sets the context for that particular pattern. After each pattern form, there is a brief two-line description of the context in which the pattern can be applied. Then the body of the problem is described in detail. The body of the problem also includes the empirical background of the pattern, sources of validity evidence, and the range of contexts in which the pattern can be applied. Then the solution part of the patterns is described, followed by a picture that indicates the pattern's main components. At end of each pattern form, all the patterns that are related to this particular pattern in the pattern language are listed. This format of pattern description is known as Alexandrian pattern form [Alexander et al. 1977]. This pattern form described patterns in a very narrative form intending to reach everyone. This pattern concept was adopted by the software community for its rather straightforward relation to the object-oriented constructs.

Patterns were widely adopted by the software community after the publication of the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich

Gamma, Richard Helm, Ralph Johnson, and John Vlissides in 1995 [Gamma et al. 1995]. This book created an important influence in the software pattern community. Since then, object-oriented design patterns have been considered to be the most popular and influential pattern work among the other patterns such as architectural patterns and idioms [Buschmann et al. 2007].

Gamma et al. captured and represented recurring design problems and solutions in traditional object-oriented programming in a domain independent way by describing them in a natural language using elements such as classes, methods, interfaces, and objects [Gamma et al. 1995]. The object-oriented patterns described by Gamma et al. are not from any specific domains. However, in recent years, catalogs of domain-specific patterns have been made available in a number of domains: parallel programming [Beverly et al. 2004], embedded systems [Konrd et al. 2004], service-oriented architectures [Endrei 2004], concurrent and distributed systems [Buschmann et al. 2007], and so forth.

Patterns have been widely used by the software community because of several reasons: they are abstractions generated from the valuable experiences of developers in solving problems that are repeatedly encountered within certain contexts, they capture design experience in such a way that they become a learning aid for novice designers, they provide a standard vocabulary among developers, and they capture the essential parts of a design in a compact form, e.g., for documenting the existing software architectures [Agerbo and Cornils 1998] [Chinnasamy 2000] [Eden 2000] [France et al. 2004] [Mikkonen 1998] [Taibi and Ling 2003 B].

Although patterns have been widely used, there is no standard definition for *what a pattern is?* There are many definitions in existence in the software engineering field for what a pattern is. One of the definitions given by Professor Christopher Alexander is "each pattern is a three-part rule that expresses a relation between a certain context, a certain system of forces that occurs repeatedly in that context, and a certain software configuration that allows these forces to resolve themselves in the software community itself" [Alexander 1979]. Another definition for patterns provided by Buschmann et al. [Buschmann et al. 1996] is "a pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution; the solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate". These definitions agree that a pattern should provide a proven solution to a recurring problem in a specific context.

At the early stages of pattern evolution, patterns were described only by using *pattern forms* [Gamma et al. 1995]. Although pattern authors tend to follow their own pattern forms, the following pattern forms are well-known and widely used by the pattern community: Alexandrian Form [Alexander et al. 1977], GoF Form [Gamma et al. 1996], POSA (Pattern-Oriented Software Architecture) Form [Buschmann et al. 1996], and Coplien Form [Coplien 1991]. All these pattern forms define the essential elements of patterns using textual descriptions, sample code fragments, and graphical modeling languages. There are a few differences in the elements that are described in the various forms. There are also some common elements, which are considered to be the essential elements of patterns, that are described by almost all the pattern forms.

What follows are brief descriptions of the essential elements of a pattern that are described in all the pattern forms [Chinnasamy 2000].

**Name:** It assigns a meaningful name to a pattern. The name is usually an abstracted representation of the participants (classes and objects involved in a pattern) and their responsibilities of the associated pattern.

**Problem:** This is the intent of the pattern, it provides a detailed description of the design problem being addressed by the pattern.

**Consequences:** This is the *responsibilities and rewards* involved in applying the pattern.

**Context:** It describes the context in which the specific design problem can recur and for which the solution is desirable. This can also be considered as a precondition to the system for which the pattern can be applied.

**Forces:** It provides descriptions on the constraints and how these constraints may conflict with the goal that can be achieved by using the pattern. Forces describe the minutiae of the constraints and solutions that can be considered in the presence of those constraints.

**Solution:** It describes the participants, i.e., classes, objects, their relationships, and their responsibilities (for object-oriented design patterns). It also adds descriptions on how to construct the participants and the relationships among the participants of the pattern. It also includes the structure of the pattern as a picture created using graphical modelling tools such as UML. The solution section is quite abstract in that it presents the solution to a wide variety of problems rather than providing a concrete solution to a specific issue. So, an object-oriented design pattern is neither a concrete design solution nor a complex domain-specific design solution. Instead, object-oriented design patterns capture general object-oriented design problems which occur in a particular context and abstract the key

aspects of the solutions to the problems to make it reusable. The solution section includes the guidelines to follow during the concrete realization of the pattern. Sometimes, the solution section also provides the alternative design options that can be considered during the concrete realization process.

**Examples:** One or more instances or the patterns are illustrated in detail in this section.

**Resulting Context:** This section outlines the resulting state of the system after the pattern has been applied, along with the forces (see entry on Forces above) that may arise from the current state of the system. This will help in deciding whether or not a specific pattern can solve a given design problem.

**Rationale:** This section provides a justification to the pattern. It provides a detailed description on how the pattern can solve a particular design issue and why the pattern is a desirable solution to a particular design problem.

**Related Patterns:** This section provides a list of patterns that are associated with this pattern. The related pattern can often be considered as a set of components that can be used to construct a larger system.

**Known Uses:** This section lists out the known occurrences of the pattern in the existing applications or systems.

As stated perviouly also, one of the widely-used pattern forms in the software pattern community is the GoF pattern form proposed by Gamma et al. [Gamma et al. 1995]. Their original design pattern form consisted of fourteen fields or sections. The GoF representation of the observer pattern is given in the following section to illustrate the discussion on the disadvantages of the textual representation of design patterns.

2.2 GoF Representation of the Observer Pattern

This section contains the representation of the observer pattern as given by Gamma et al. [Gamma et al. 1995].

**Intent**

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

**Also Known As**

Dependents, Publish-Subscribe

**Motivation**

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency among related objects. Consistency is not to be achieved by making the classes tightly coupled, because that reduces their reusability. For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data. Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and a bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they behave as though they do. When a user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa. This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there is no reason to limit the number of dependent objects to two, there may be any number of different user interfaces to the

same data.  The Observer pattern describes how to establish these relationships. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. This kind of interaction is also known as publish-subscribe. The subject is the publisher of the notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

**Applicability**

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets the users vary and reuse them independently.

- When a change to one object requires changing others, and there is no telling how many objects need to be changed.

- When an object should be able to notify other objects without making assumptions about the identity of these objects. In other words, it is not desirable for these objects to be tightly coupled.

**Participants**

- Subject
  - Knows its observers. Any number of Observer objects may observe a subject.
  - Provides an interface for attaching and detaching Observer objects.

- Observer
  - Defines an updating interface for objects that should be notified of changes in a subject.

- Concrete Subject

  o Stores the state of interest to Concrete Observer objects.

  o Sends a notification to its observers when its state changes.

- Concrete Observer

  o Maintains a reference to a Concrete Subject object.

  o Stores the state that should stay consistent with the subject's state.

  o Implements the Observer updating interface to keep its state consistent with the subject's state.

**Structure**
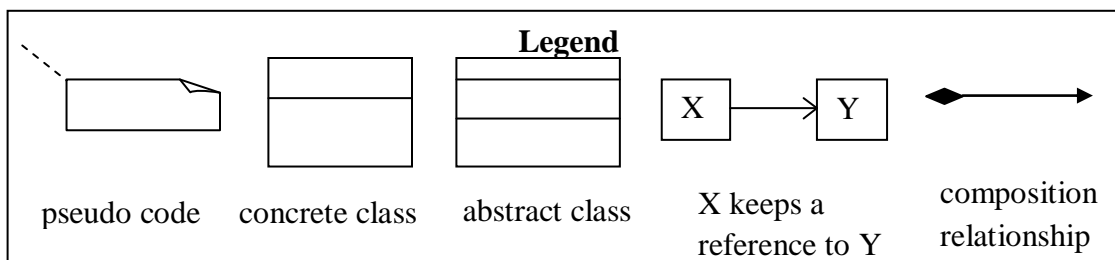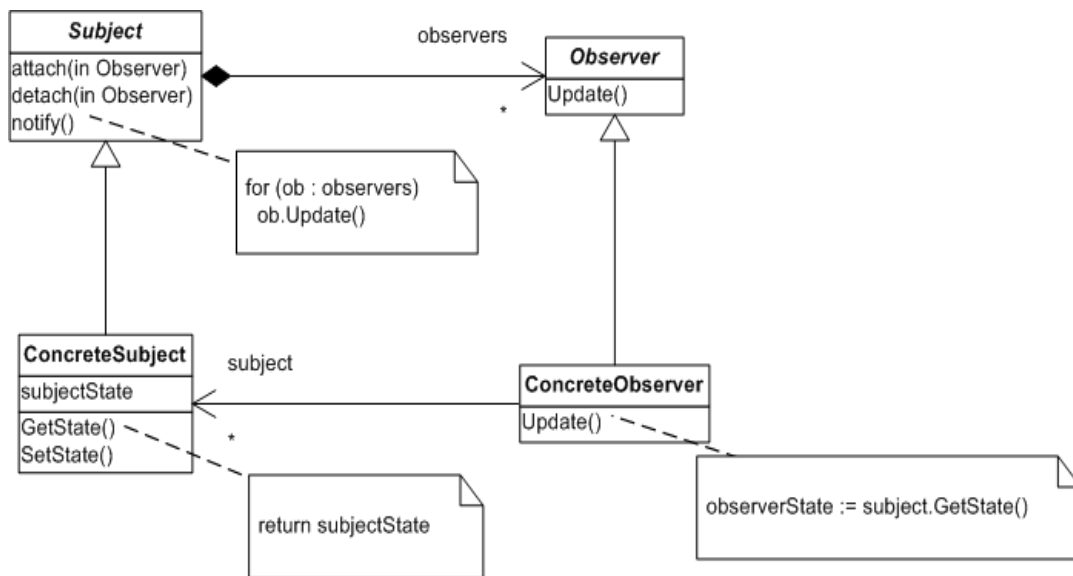


Figure 2.1 UML Representation of the Observer Pattern

17

**Collaborations**

- Concrete Subject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.

- After being informed of a change in the concrete subject, a Concrete Observer object may query the subject for information. Concrete Observer uses this information to reconcile its state with that of the subject. The following interaction diagram illustrates the collaborations between a subject and two observers. Interaction diagrams model the behavior of use cases by describing the way groups of objects interact to complete a task [Fowler 2003].

- Note how the Observer object that initiates the change request postpones its update until it gets a notification from the subject. Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely. The Implementation section discusses some common variations.

**Consequences**

The Observer pattern lets the users vary subjects and observers independently. Users can reuse subjects without reusing their observers, and vice versa. It lets users add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following: 1. Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class. The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal. Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system.
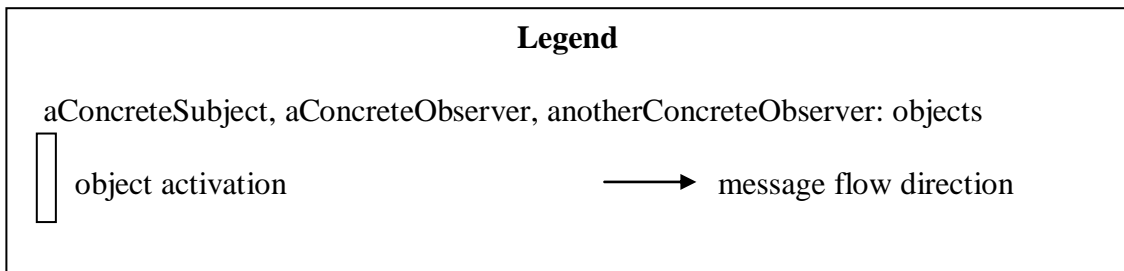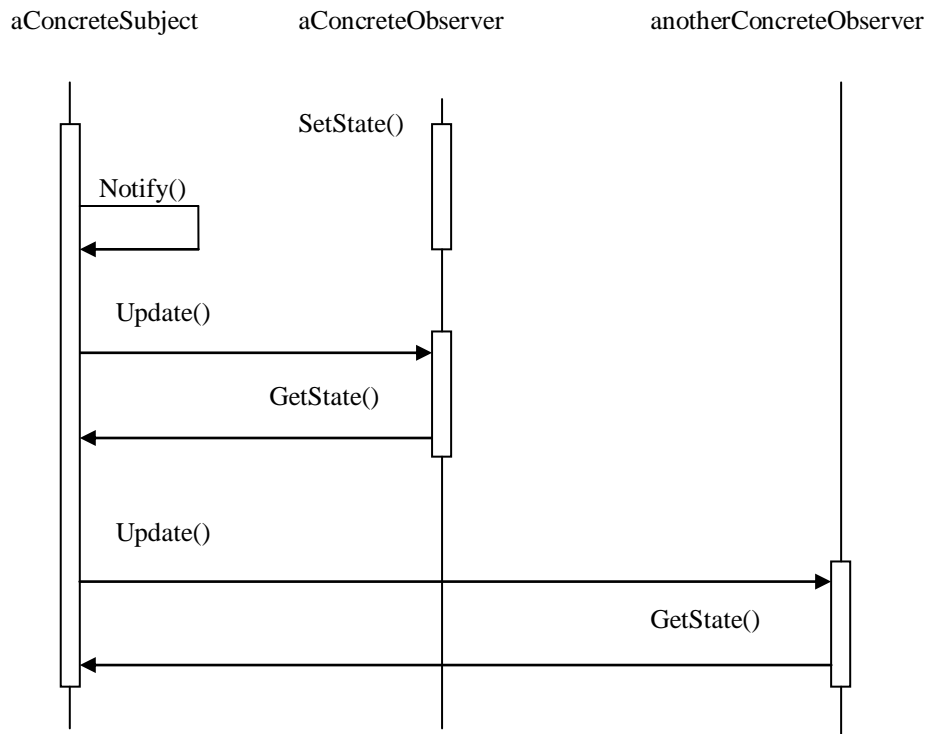
Figure 2.2 An interaction diagram depicting the collaborations of the Observer Pattern

A lower-level subject can communicate and inform a higher-level observer, thereby keeping the system's layering intact. If Subject and Observer are lumped together, the resulting object must either span two layers (and violate the layering), or it must be forced to live in one layer or the other (which might compromise the layering abstraction). Support for broadcast communication. Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast

19

automatically to all interested objects that subscribed to it. The subject doesn't care how many interested objects exist, its only responsibility is to notify its observers. This gives the users the freedom to add and remove observers at any time. It is up to the observer to handle or ignore a notification.

Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of 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 aren't well-defined or maintained usually lead to spurious updates, which can be hard to track down.

This problem is aggravated by the fact that the simple update protocol provides no details as to what changed in the subject. Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

**Implementation**

Several issues related to the implementation of the dependency mechanism are discussed in this section.

1. Mapping subjects to their observers. The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject. However, such storage may be too expensive when there are many subjects and few observers. One solution is to trade space for time by using an associative look-up (e.g., a hash table) to maintain the subject-to-observer mapping. Thus a subject with no observers does not incur any storage overhead. On the other hand, this approach increases the cost of accessing the observers.

2. Observing more than one subject. It might make sense in some situations for an observer to depend on more than one subject. For example, a spreadsheet may depend on more than one data source. It is necessary to extend the Update interface in such cases to let the observer know *which* subject is sending the notification. The subject can simply pass itself as a parameter in the Update operation, thereby letting the observer knows which subject to examine.

3. *Who triggers the update?* The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls Notify to trigger the update? Here are two options:

   a. Have state-setting operations on Subject call Notify after they change the subject's state. The advantage of this approach is that clients don't have to remember to call Notify on the subject. The disadvantage is that several consecutive operations will cause several consecutive updates, which may be inefficient.

   b. Make clients responsible for calling Notify at the right time. The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, thereby avoiding needless intermediate updates. The disadvantage is that clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

4. *Dangling references to deleted subjects.* Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to

it. In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

5. *Making sure the Subject state is self-consistent before notification.* It is important to make sure the Subject state is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state. This self-consistency rule is easy to violate unintentionally when the Subject subclass operations call inherited operations.

6. Avoiding observer-specific update protocols: the push and pull models. Implementations of the Observer pattern often have the subject broadcast additional information about the change. The subject passes this information as an argument to Update. The amount of information may vary widely. At one extreme, which we call the push model, the subject sends observers detailed information about the change, whether they want it or not. At the other extreme is the pull model: the subject sends nothing but the most minimal notification and the observers ask for details explicitly thereafter. The pull model emphasizes the subject's ignorance of its observers, whereas the push model assumes that the subjects know something about their observers' needs. The push model might make observers less reusable, because Subject classes make assumptions about Observer classes that might not always be true. On the other hand, the pull model may be inefficient, because Observer classes must ascertain what changed without help from the Subject.

7. Specifying modifications of interest explicitly. Update efficiency can be improved by extending the subject's registration interface to allow registering observers only for specific events of interest. When such an event occurs, the subject informs only those

observers that have registered their interest in that event. One way to support this uses the notion of aspects for Subject objects. To register interest in particular events, observers are attached to their subjects using

> a. void Subject::Attach(Observer*, Aspect& interest);

where "interest" specifies the event of interest. At notification time, the subject supplies the changed aspect to its observers as a parameter to the Update operation. For example:

> b. void Observer::Update(Subject*, Aspect& interest);

8. Encapsulating complex update semantics. When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. Such an object is called a ChangeManager. Its purpose is to minimize the work required to make observers reflect a change in their subject. For example, if an operation involves changes to several interdependent subjects, it might have to be ensured that their observers are notified only after all the subjects have been modified to avoid notifying observers more than once. ChangeManager has three responsibilities:

   a. It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.

   b. It defines a particular update strategy.

   c. It updates all dependent observers at the request of a subject.

The following diagram depicts a simple ChangeManager-based implementation of the Observer pattern. There are two specialized ChangeManagers. The

SimpleChangeManager is naive in that it always updates all observers of each subject. In contrast, the DAGChangeManager handles directed-acyclic graphs of dependencies among subjects and their observers.

A DAGChangeManager is preferable to a SimpleChangeManager when an observer observes more than one subject. When more than one subject is observed by an observer, a change in two or more subjects might cause redundant updates. The DAGChangeManager ensures that the observer receives just one update. The SimpleChangeManager is fine when multiple updates do not pose an issue.
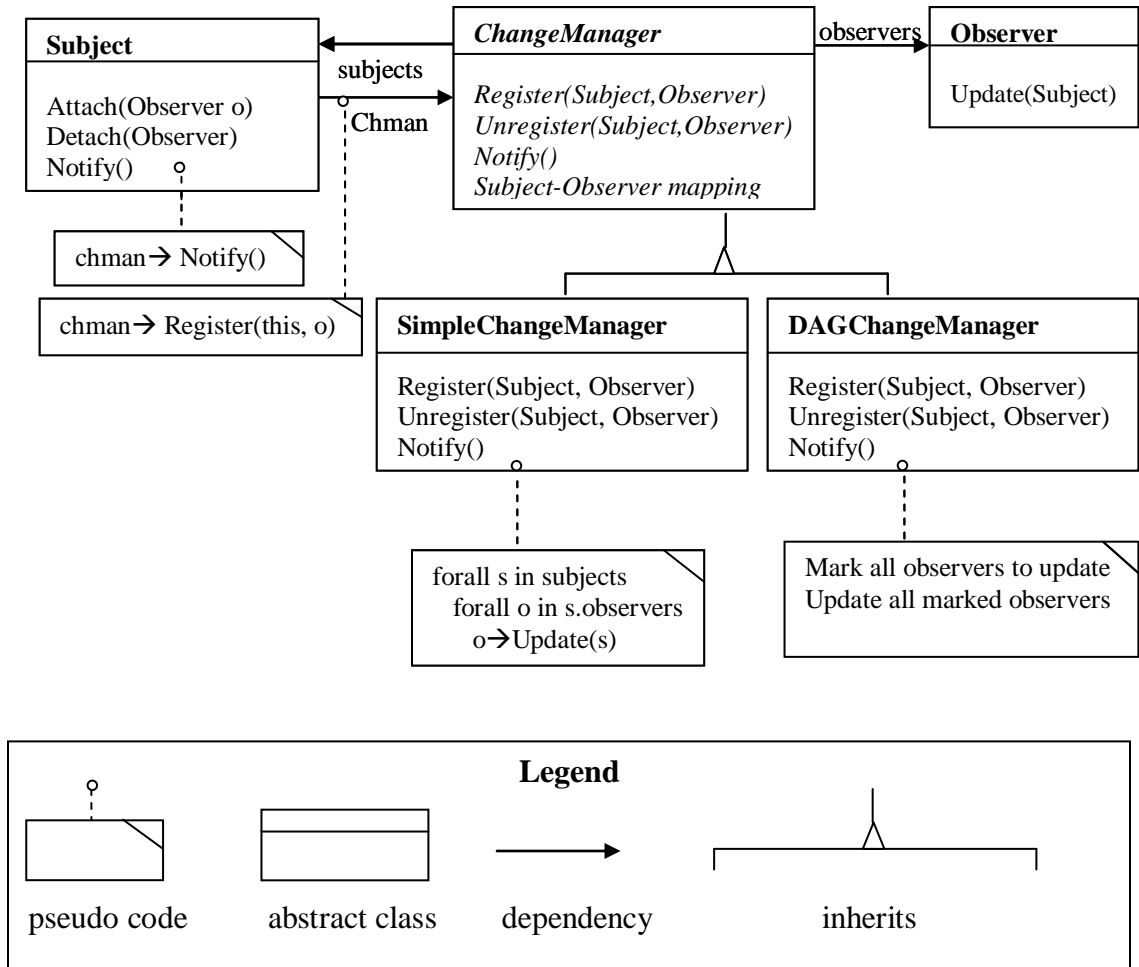


Figure 2.3 UML Representation of an Instance of the Observer Pattern

9. Combining the Subject and Observer classes. Class libraries written in languages that lack multiple inheritance (e.g., Smalltalk) generally don't define separate Subject and Observer classes but combine their interfaces in one class. That lets the users define an object that acts as both a subject and an observer without multiple inheritance. In Smalltalk, for example, the Subject and Observer interfaces are defined in the root class Object, making them available to all classes.

**Sample Code of the Observer Pattern**

An abstract class defines the Observer interface:

```
class Subject;

class Observer {
public:
        virtual ~ Observer();
        virtual void Update(Subject* theChangedSubject) = 0;
protected:
        Observer();
};
```

This implementation supports multiple subjects for each observer. The subject passed to the Update operation lets the observer determine which subject's state is changed when the observer is observing more than one subject.

Similarly, an abstract class defines the Subject interface:

```
class Subject {
public:
        virtual ~Subject();
        virtual void Attach(Observer*);
        virtual void Detach(Observer*);
        virtual void Notify();

protected:
        Subject();
        private: List<Observer*> *_observers;
};
```

25

```
void Subject::Attach (Observer* o) {
        _observers->Append(o);
}

void Subject::Detach (Observer* o) {
        _observers->Remove(o);
}

void Subject::Notify () {
        ListIterator<Observer*> i(_observers);
        for (i.First(); !i.IsDone(); i.Next()) {  i.CurrentItem()->Update(this); }
}
```

ClockTimer is a concrete subject for storing and maintaining the time of day. It notifies

its observers every second. ClockTimer provides the interface for retrieving individual

time units such as the hour, minute, and second.

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick();
};
```

The Tick operation gets called by an internal timer at regular intervals to provide an

accurate time base. Tick updates the ClockTimer's internal state and calls Notify to

inform observers of the change:

```
void ClockTimer::Tick () {
        // update internal time-keeping state  // ...  Notify();
}
```

Now we can define a class DigitalClock that displays the time. It inherits its graphical

functionality from a Widget class provided by a user interface toolkit. The Observer

interface is mixed into the DigitalClock interface by inheriting from Observer.

```
class DigitalClock: public Widget, public Observer {
public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();
        virtual void Update(Subject*);   // overrides Observer operation
        virtual void Draw();                  // overrides Widget operation;
                                              // defines how to draw the digital clock
private:
        ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
_subject = s;  _subject->Attach(this);
}

DigitalClock:: DigitalClock () {  _subject->Detach(this); }
```

Before the Update operation draws the clock face, it checks to make sure the notifying

subject is the clock's subject:

```
void DigitalClock::Update (Subject* theChangedSubject) {
        if (theChangedSubject == _subject) {   Draw();  }
}

void DigitalClock::Draw () {
        // get the new values from the subject
        int hour = _subject->GetHour();
        int minute = _subject->GetMinute();        // draw the digital clock
}
```

An AnalogClock class can be defined in the same way.

```
class AnalogClock : public Widget, public Observer {
public:
        AnalogClock(ClockTimer*);
        virtual void Update(Subject*);
        virtual void Draw();  // ...
};
```

The following code creates an AnalogClock and a DigitalClock that always show the

same time:

```
ClockTimer* timer = new ClockTimer;
```

AnalogClock\* analogClock = new AnalogClock(timer);

DigitalClock\* digitalClock = new DigitalClock(timer);

Whenever the timer ticks, the two clocks will be updated and will redisplay themselves appropriately.

**Known Uses**

The first and perhaps best-known example of the Observer pattern appears in Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment [Krasner and Pope 1988]. MVC's Model class plays the role of Subject, while View is the base class for observers. Smalltalk, ET++, and the THINK class library provide a general dependency mechanism by putting Subject and Observer interfaces in the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern are InterViews, the AndrewToolkit, and Unidraw [Kvale 1996] [Palay et al. 1988] [Vlissides and Linton 1990]. InterViews defines Observer and Observable (for subjects) classes explicitly. Andrew calls them "view" and "data object", respectively. Unidraw splits graphical editor objects into View (for observers) and Subject parts.

**Related Patterns**

Mediator: By encapsulating complex update semantics, the ChangeManager acts as mediator between subjects and observers.

Singleton: The ChangeManager may use the Singleton pattern to make it unique and globally accessible.

2.2.1 Deficiencies in the GoF Representation

The textual description provided by the GoF pattern form [Gamma et al. 1995] is quite detailed and is a suitable way to communicate design knowledge in a domain independent way. Although examples improve understanding, the lengthy description (around 10 pages for each pattern) spread out over multiple sections, thus making it rather challenging for a developer to get the core of the pattern because the implementation detail is sometimes lost in the long-winded description [Sabatucci et al. 2009]. Pattern users are expected to enhance their design experience by studying these design patterns and use their pattern knowledge in the development process. These patterns are represented at a level of abstraction and in a natural language mode that requires human interpretation of the pattern contents to the desired implementation. The natural language descriptions restrict precise interpretation of the design patterns also hinder tool support and automation.

Henninger and Corrêa collected a number of patterns published in the 1994-2007 time frame to show the pros and cons of the increase in the number of patterns being published, and analyze the trend in pattern practice [Henninger and Corrêa 2007]. They collected and analyzed 2241 software design patterns from various resources including the GoF book [Gammat et al. 1995], PLoP (Pattern Languages of Programs) proceedings [Hillside 1993], POSA (Pattern-Oriented Software Architecture) [Buschmann et al. 1996], and Fowler's book titled Analysis Patterns [Fowler 1997]. They advance the argument that their collection of patterns should be considered an underestimate of the actual number of patterns available, as "it is a daunting task to find all patterns in various printed and electronic sources" [Henninger and Corrêa 2007].

Even before 2000, when the number of patterns being published was reaching over 1000 [Rising 2000], there was discussion on the manageability of the increasing number of available patterns: "...there are now so many patterns it is very difficult to remember them all" [Cline 1996] and that "the increase in the number of design patterns makes a common vocabulary unmanageable" [Agerbo and Cornils 1998]. Since 2000, the number of patterns published has more than doubled and the increase is probably due to the diverse set of software systems being developed [Henninger and Corrêa 2007].

The advantage of such an increase in the number of patterns is that "the body of knowledge collectively represented by patterns is vast and increasing" [Henninger and Corrêa 2007]. But the drawback is that the number of patterns published is increasing and it could reach a point where it would become infeasible to identify all potentially relevant patterns to a specific situation [Henninger and Corrêa 2007] [Buschmann et al. 2007] [Chinnasamy 2000] [Soundarajan and Hallstrom 2004] [Taibi and Ling 2003 B]. Therefore, the need for tool support is becoming critical. The natural language representation of patterns restricts the development of automated tools support due to the inherently informal semantic description of natural languages. In spite of the increasing number of available patterns, duplicate patterns, and the lack of a standard medium for communicating software patterns, they have the great potential to become a unique medium for capturing and communicating domain knowledge about the best practices of software development.

The motivation behind the natural description of the patterns is to communicate design knowledge in a domain independent way [Buschmann et al. 2007]. But natural language representation has serious flaws. The pattern users are faced with difficulties in

understanding when and how to use the increasing number of available patterns [Taibi and Ling 2003 B] [Bayley and Zhu 2008]. This is mainly due to the natural language textual description of patterns which is informal, ambiguous, and sometimes misleading in attempting to understand and apply them [Taibi 2006] [Bayley and Zhu 2008]. As a result, pattern users were forced to understand the meaning of patterns from their interpretation of the patterns' verbal specification [Eden 2000]. Hence, there was a need felt to formalize patterns in order to describe them accurately, to reason about them rigorously, and also to facilitate tool support [Agerbo and Cornils 1998] [Bayley and Zhu 2008] [Buschmann et al. 2007] [Chinnasamy 2000] [Dong 2002] [Eden 2000] [France et al. 2004] [Kent and Lauder 2004] [Mikkonen 1998] [Soundarajan and Hallstrom 2004] [Taibi and Ling 2003 B].

Formal specification approaches were used in an attempt to formalize patterns using existing formal specification languages, programming languages, object-oriented notations, as well as by devising special purpose notations and specification languages. For example, Bosch [Bosch 1996] and Dong [Dong 2002] formalized design patterns using C++ and UML, respectively. Mikkonen [Mikkonen 1998], Eden and Hirshfeld [Eden and Hirshfeld 2001], and Chinnasamy [Chinnasamy 2000] derived special purpose specification notations such as Distributed Coordination (DisCo), LanguagE for Pattern Uniform Specification 3 (LePUS3), and Extended LanguagE for Pattern Uniform Specification (eLePUS), respectively. Eden and Hirshfeld [Eden and Hirshfeld 2001] and Taibi and Ling [Taibi and Ling 2003 B] devised special purpose specification languages called Balanced Pattern Specification Language (BPSL) and class-Z.

Among the existing formal specification schemes, DisCo, BPSL, and LePUS3 are considered to be the most promising specification languages for design patterns [Taibi 2007] [Henninger and Corrêa 2007] [Mak et al. 2004]. The rest of this section contains a brief description of these three schemes. Sections 2.3, 2.4, and 2.5 discuss these three schemes in more detail followed by their comparison in Section 2.6.

DisCo is a formal specification method proposed by Mikkonen [Mikkonen 1998]. In DisCo, the specifications and the modeling of the interactions are done at a high level of abstraction. The formal basis of this method is the Temporal Logic of Actions. DisCO was mainly aimed at formalizing the behavioral aspects of design patterns, hence its specification of the structural aspects of the patterns is not as good compared to the specification of the behavioral aspects of patterns. Also, DisCo did not provide a repository management system for patterns.

Taibi [Taibi 2007] proposed the Balanced Pattern Specification Language (BPSL) that uses First Order Logic and Temporal Logic of Actions to formalize the static and dynamic (or structural and behavioral) aspects of patterns. This specification scheme uses mathematical notations to formalize patterns. As its name suggests, BPSL attempts to provide a balanced specification of the structural and behavioral aspects of design patterns, but fails to concentrate on the understandability of the resulting specification. Pattern specification schemes/languages should be understandable and not unduly complex to the users of patterns [Kim and Carrington 2004]. The mathematical notations used for the formal specification schemes should not be a burden for the developer. The developers' having to delve into the details of the mathematical foundations of a

formalism, before they can understand the meaning of a pattern, constitutes extra and arguably unnecessary work for pattern users.

One of the notable works on formal specification of design patterns is LePUS3 and Class-Z [Eden and Hirshfeld 2001], with LePUS3 being an extension of LePUS. LePUS is graphical formal specification language. This language expresses the relationships among different patterns and the relationships among the different elements of a pattern, using a simplified Higher Order Monadic Logic formalism. Since LePUS did not provide the means to capture the behavioral aspects of patterns, Chinnasamy [Chinnasamy 2000] extended LePUS and provided eLePUS which can be used to capture the structural as well as the behaviroal aspects of design patterns. LePUS was also extended by Eden and Hirshfeld [Eden and Hirshfeld 2001] to represent both structural and behavioral aspects of patterns. LePUS3 mainly represents the relationships that exist among the participants of design patterns. The visual notations of LePUS3 are supported by the formal specification language Class-Z [Eden et al. 2007]. Class-Z is derived from the formal specification language Z [Eden et al. 2007]. LePUS3 and Class-Z can be used to capture the static and behavioral aspects of design patterns. It seems that LePUS3 is still in its initial evolutionary phases. This two-tier programming tool support for the specification, verification, and visualization of design patterns as well as software systems is still a work in progress [Eden and Gasparis 2009]. One of the main limitations of LePUS3 and Class-Z is that they do not really capture or represent the behavioral elements of design patterns accurately [Eden et al. 2007].

The following sections provide detailed descriptions of DisCo, BPSL, and LePUS3 & Class-Z.

## 2.3 Distributed Cooperation (DisCo)

### 2.3.1 Introduction

DisCo is an object-oriented specification language for specifying the behavioral aspects of reactive systems proposed by Mikkonnen [Mikkonen 1998]. A reactive system is "one that is in continuous interaction with its environment, this is in contrast to the transformational theories, where a system is understood to transform input into output" [Mikkonen 1995]. In DisCo, the specification and modeling of interactions are done at a high level of abstraction. The formal basis of this method is the *Temporal Logic of Actions*.

### 2.3.2 Building Blocks of DisCO

The building blocks of DisCo are: *classes, relations,* and *actions* [Mikkonen 1995]. *Classes* are structures that describe the elements of an *object*. An *Object* in DisCo is a structure that may contains: *state machines*, *variables*, and *references to other objects*. State machines represent different states of the system. Variables are of basic types such as integer, string, and character. State machines, variables, and references together represent the current or *local state* of the object. *Global state* of the system can be determined by combining the local state of all the objects. *Relations* are used to associate objects with other objects. Relations are defined in the following format:

**relation** (n).R.(m): *C* x *D*

Here, relation *R* associates n instances of class C with m instances of class D [Mikkonen 1998].

Actions in DisCo are atomic events that are executed in an interleaving manner and are selected non-determinately, i.e., if alternative actions are possible at the same time, the selection between them is nondeterministic (one action among other actions is selected without any specification of which one will be taken). The specification of an action contains three parts: a header, a guard, and a body. The header specifies the name of the action as well as a list of participating objects and parameters. The guard is a precondition, which is boolean expression is upon the satisfaction of the guard, the corresponding action is executed by the system. The body of an action contains a set of statements that might change the state of the participating objects when the action is executed.

A simple DisCo class [Mikkonen 1995]:

```
class sample = {
        state *a, b; //a state machine; * indicates the default state of the state
machine
        value: integer:= 0; //an integer variable
}
```

A sample DisCo action [Mikkonen 1995]:

```
Act (s: sample; value)                         //header
        when s.a do                            //guard
                → s.b;                         //body, a state transition
                s.value := s.value + 1;        //an assignment
```
The action *Act,* specified above, can be invoked only when there is an instance *s* of a class called *sample* and the *local state* of its *state machine* is *a*. When the action *Act* executes, it changes the local state of state machine of class *sample* to *b* and increments the variable *value* of the class *sample* by one.

In addition to specifying the bahaviroal aspects of a system, DisCo also provides methods to modularise DisCo specifications by refining them. Refinements are applied

by making sure the *safe properties hold (nothing bad will ever happen)* using *proof obligations*. Each refinement that is applied to the specification modularizes the behaviroal aspect of the system as a whole rather than modularizing the local behavior of a participant in the system. During this refinement process, new classes and variable are added, and the exisitng actions are refined to reflect the changes made to the newly added classes and variables. Refinements to the actions are applied by inheriting the existing actions.

2.3.3 DisCo Representation of the Observer Pattern

This subsection contains the representation of the observer pattern in the DisCo specification followed by a brief discussion on the relations involved in the specification.

**class** *Subject = { Data }*
**class** *Observer = { Data }*

//asterisk (*) stands for any possible number of instances

**relation** *(0..1).Attached(\*): Subject* x *Observer*
**relation** *(0..1).Updated(\*): Subject* x *Observer*

*Attach(s: Subject; o: Observer)*:
  ¬ *s.Attached.o*   //¬ represents logical NOT, i.e., o is not attached to s
  ➔ *s.Attached′.o*   //➔ represents state transition

*Detach(s: Subject; o: Observer)*:
  ¬ *s.Attached.o*
  ➔ ¬ *s.Attached′.o*
  ∧¬ *s.Updated′.o*   //∧ represents logical AND


*Notify(s: Subject, d)*:
  ➔ ¬ *s.Updated′.***class** *Observer*
  ∧ *s.Data′ = d*

*Update(s: Subject; o\*: Observer; d)*:

$$s.Attached.o$$
$$\wedge \neg\ s.Updated.o$$
$$\wedge\ d = s.Data$$
$$\wedge\ s.Updated'.o$$
$$\wedge\ o.Data' = d$$

DisCo specification introduced two relations: *Attached* and *Updated*. The relation *Attached* is defined over a subject and a set of observers. This relation captures the *Observers* that are currently attached to the *Subject*. An observer can be attached to the subject whenever the action *Attach()* is executed. *Attach()* can only be executed when there exist instance *s* of a class *Subject* and another instance *o* of a class *Observer*. An observer is disassociated from the subject when the *Detach()* action is executed. The relation *Updated()* is defined over a subject and a set of observers. It is used to capture the observers that were updated by the subject after the *Notify()* action was last executed [Mikkonen 1998].

## 2.4 Balanced Pattern Specification Language (BPSL)

2.4.1 Introduction

BPSL [Taibi and Ling 2003 B] is an attempt to formalize both the structural and behavioral aspects of design patterns using a subset of First Order Logic (FOL) and a subset of Temporal Logic of Actions (TLA). The subset of TLA is used to perform actions such as changing state variables (class attributes) and associating or disassociating object with the other participants of a design pattern. This work can be considered an extension to DisCo [Mikkonen 1998] and LePUS [Eden 2000] [Taibi and Ling 2003 B] [Hallstrom 2004]. In BPSL, the structural aspects of patterns are represented by the classes associated with the patterns and the relationships among them.

Classes are represented by the instances of the objects of the associated classes. The relationship between the participating classes and objects are represented as mathematical association between them using temporal relations. The behavioral aspects of patterns are captured using *actions*, which define the state changes of the associated participants.

2.4.2 Building Blocks of BPSL

The building blocks of BPSL specifications are: *entities*, *relations*, and *actions*. They are explained in the following paragraphs.

Entities include classes, attributes, methods, objects, and untyped variables. The entities are represented using the following symbols respectively *C, A, M, O, and V*. Untyped variable are variables of any type, such as a combination of a class and objects, which are used to create higher levels of abstraction [Taibi and Ling 2003 B].

*Relations* define the way entities cooperate with one another. There are two types of relations: *permanent* and *temporal*. As the name suggests, permanent relations once defined and cannot be changed, but the temporal relations change dynamically throughout the execution of actions. A temporal relation is defined as follows: TR(C1[n], C2[m]), where n and m are cardinalities. The relation TR is associated with *n* instances of class C1 and *m* instances of C2. In BPSL, the cardinality of a class can be specified either as a closed interval [n..m] or as [*]. In a closed interval ([n..m]), n and m represent any two positive integers, and [*] depicts any possible number of instances of a class [Taibi and Taibi 2006].

An action in BPSL is similar to an action in DisCo. Actions use temporal relations to either associate with or disassociate from objects from the other participants of the

patterns. Objects are the instances of classes as defined in any object-oriented software system. An action consists of a set of parameters, a precondition, and a body. Parameters of the actions can be of two types: untyped values and objects. The pre-condition of an action should be satisfied in order to execute the body of the action. The body defines the state changes caused by an execution of the action. Actions are atomic and selected non-deterministically. Temporal relations are used in the actions to associate with and disassociate from objects from the other participants of the patterns. For example, the temporal relation TR(Object1,Object2) indicates that an object Object1 of a class C1 is currently associated with an object Object2 of a class C2 through TR, while the relation ¬TR(Object1, Object2) shows that Object1 and Object2 are no longer associated through TR [Taibi and Taibi 2006].

2.4.3 Representation of Structural Aspects in BPSL

A subset of First Order Logic (FOL) is used to define the structural aspects of design patterns in BPSL. Structural aspects are represented as expressions that use logical connectives (mainly $\vee$ (logical or) and $\wedge$ (logical and)), quantifiers (mainly$\exists$ ), and predicate symbols to impose constraints on the variable symbols. Variable symbols are the primary entities of BPSL. A predicate is a Boolean expression defined over the variables and classes of the system. Predicates define *permanent relations* among the entities. These relations are derived from the object-oriented concepts.

The relations described in Table 2.1 are the primary permanent relations, their domain, and their intent. Other permanent relations can be derived from these primary

39

permanent relations. Primary permanent relations are extracted from the object-oriented

concept [Taibi and Taibi 2006].

| Name | Domain | Intent |
|---|---|---|
| *Defined-in* | *MxC* | Indicates that a method is defined in a certain class. |
| | *AxC* | Indicates that an attribute is defined in a certain class. |
| *Reference-to-one (-many)* | *CxC* | Indicates that one class defines a member whose type is a reference to one (many) instance(s) of the second class. |
| *Inheritance* | *CxC* | Indicates that the first class inherits from the second. |
| *Creation* | *MxC* | Indicates that a method contains an instruction that creates a new instance of a class. |
| | *CxC* | Indicates that one of the methods of a class contains an instruction that creates a new instance of another class. |
| *Invocation* | *MxM* | Indicates that the first method invokes the second method. |
| *Argument* | *CxM* | Indicates that a reference to a class is an argument of a method. |
| | *VxM* | Indicates that an untyped value is an argument of a method. |
| *Instance* | *OxC* | Indicates that an object is an instance of a certain class. |

Table 2.1 Primary permanent relations, their domains, and their intents in BPSL [Taibi

and Taibi 2006]

2.4.4 Representation of Behavioral Aspects

The behavioral aspects of patterns are specified using a subset of TLA (Temporal

Logic of Actions). The behavioral aspects of a pattern are represented as the consecutive

state changes resulting from the execution of consecutive actions. These sequences of

state changes can be potentially infinite. Each state represents the values of its state

40

variables and the temporal relations among the objects. State variables are the attributes of a class in a particular state. Actions are selected dynamically and are executed when the preconditions of the actions are satisfied. As mentioned above (see Section 2.4.2), actions change the state of system and associate objects with or disassociate objects from the other entities such as classes, attributes, methods, objects, and untyped variables of the system. The system will start in some initial state. As a side effect of the actions and the execution on the system, the state of the system will change accordingly.

2.4.5 Representation of the Observer Pattern in BPSL

This subsection contains the representation of the observer pattern in the BPSL schema followed by a brief explanation of the entities, relations, and actions involved in the schema. Figure 2.4 contains the representations of the observer pattern in BPSL (see Section 2.4.2 for a detailed description of the symbols used in the BPSL specification of the observer pattern).

Patterns are represented as formulas in BPSL. Both permanent and temporal relations are expressed as formulas. The Observer pattern specification has two temporal relations and six types of primary permanent relations (see Table 2.1 for a list of the primary permanent relations and their definitions in BPSL). *Attached* and *Updated* are the two temporal relations. *Defined-in, Inheritance, Invocation, Argument, Reference-to-one,* and *Reference-to-many* are the six types of primary permanent relations used in the BPSL specification of the observer pattern.

$\exists$ *subject, concrete-subject, observer, concrete-observer* $\in C$;
*subject-state, observer-state* $\in A$;
*attach, detach, notify, get-state, set-state, update* $\in M$;

41

*Defined-in(subject-state, concrete-subject) ∧*
*Defined-in (observer-state, concrete-observer) ∧*
*Defined-in(attach, subject) ∧*
*Defined-in (detach, subject) ∧*
*Defined-in (notify, subject) ∧*
*Defined-in(set-state, concrete-subject) ∧*
*Defined-in (get-state, concrete-subject) ∧*
*Defined-in (update, observer) ∧*
*Reference-to-one(concrete-observer, concrete-subject) ∧*
*Reference-to-many(subject, observer) ∧*
*Inheritance(concrete-subject, subject) ∧*
*Inheritance(concrete- observer, observer) ∧*
*Invocation(set-state, notify) ∧*
*Invocation(notify, update) ∧*
*Invocation(update, get-state) ∧*
*Argument(observer, attach) ∧*
*Argument(observer, detach) ∧*
*Argument(subject, update)*

*Attached (concrete-subject<0..1>,concrete-observer<[\*]>,*
*Updated (concrete-subject<0..1>,concrete-observer<[\*]>) ∈ TR;*

*s ,o ∈ O;s ∈ concrete-subject; o ∈ concrete-observer; d∈ V;*
*Initially: ¬Attached(s, concrete-observer)*
*Attach(s, o): ¬Attached(s, o) → Attached'(s, o)*
*Detach(s, o): Attached(s, o) ∨ (Attached(s, o) ∧ Updated(s, o)) →*
　　　　　　*¬Attached'(s, o) ∧¬Updated'(s, o)*
*Notify(s,o,d) : Attached(s,o) ∨ (Attached(s, o) ∧ Updated(s, o)) →*
　　　　　　*¬Updated'(s, concrete-observer) ∧　((s.subject-state)'=d)*
*Update\*(s, o):　Attached (s, o) ∧ ¬Updated(s, o) → Updated'(s, o) ∧ (*
　　　　　　*(o.observer-state)' = s.subject-state)*

Figure 2.4 Representation of the Observer Pattern in BPSL [Taibi and Taibi 2006]

The *Temporal Relations (TR) Attached* and *Updated* are defined over a subject

and a set of observers. The term *Initially* defines the initial state of the system. *Attach,*

*Detach, Notify*, and *Update* are the actions defined in the observer pattern. *Attach*

associate an observer with a subject and *Detach* disassociate an observer from a subject.

*Notify* indicates that the state of a concrete subject have been modified thus initiating the

update process to update all the concrete observers that are associated with the concrete subject.

### 2.5 Language for Uniform Pattern Specification3 (LePUS3) and Class-Z

2.5.1 Introduction

LanguagE for Pattern Uniform Specification 3 (LePUS3) and Class-Z are object-oriented Design Description Languages (DDL) that are intended to abstract, model, and formalize object-oriented programs, design patterns, and application frameworks. LePUS3 is an extension of LePUS [Eden 2000]. LePUS3 and Class-Z are defined using first-order predicate calculus. The semantics of LePUS3 and Class-Z specifications were defined using the standard language of mathematical logic including model theory, predicate calculus, and elementary set theory. The semantics of LePUS3 and Class-Z are specifications of the *abstract semantics* of programs that is an abstract representation of programs written in object-oriented languages. Therefore the atomic units of LePUS3 and Class-Z specifications are the basic elements of object-oriented programs. These elements are classes, methods, method signatures, and relationships such as *inherits from*, *defined in*, and *creates instances of* that exist among the classes and methods. In the LePUS3 and Class-Z notation, classes and methods are known as *entities*.

2.5.2 Building Blocks of LePUS3 and Class-Z

LePUS3 uses visual notations to model object-oriented units (i.e., classes and methods) whereas Class-Z is a symbolic language which is basically an extension of the formal specification language Z [Spivey 1992]. For example, LePUS3 uses a rectangle to represent a class and a shaded rectangle to represent a set of classes. Appendix B lists the

symbols, terms, and relations used in LePUS3 and Class-Z. Specifications represented in

LePUS3 can be represented in Class-Z, and vice versa. A specification is either a

*Codechart* expressed in LePUS3 or a schema expressed in Class-Z. The structure of a

Class-Z schema is displayed in Figure 2.5. *Codechart* is the visual notation expressed in

LePUS3. Codechart of the observer pattern is given in Figure 2.6 and the representation

of the observer pattern in Class-Z is given in Figure 2.7. Specifications written in

LePUS3 and Class-Z are *formulas* that use *predicate* and *relation symbols* to express the

constraints of the participating *entities* and the *relationships* that exist among the entities.

Appendix B lists all the predicate and relation symbols in Class-Z schema.

A Class-Z Schema is a specification with a specific format given below.

```
┌─ Schema Name ──────────────────────────────
│   declaration : TYPE
│   declaration : TYPE …
├─────────────────────────────────────────────
│
│   formula
│   formula
│   …
└─────────────────────────────────────────────
```
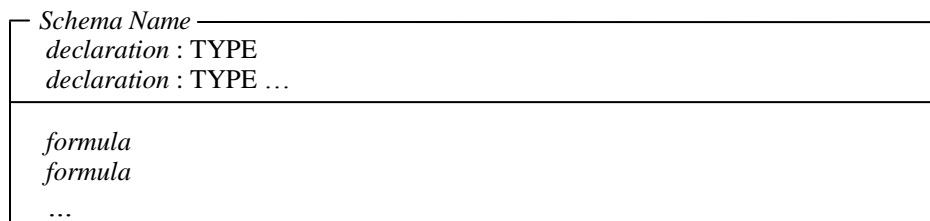
Figure 2.5 Class-Z Schema [Eden et al. 2007]

Three participants of the observer pattern are represented in the codechart

representation (see Figure 2.6): *Subject*, which is an abstract class; *concreteSubject*,

which is a class; and *observers*, which is an inheritance variable. The inheritance variable

*observers* can have any number of class variables that can inherit from the *observers*

hierarchy variable. Members of each of the three participants are given in elliptical

shapes.
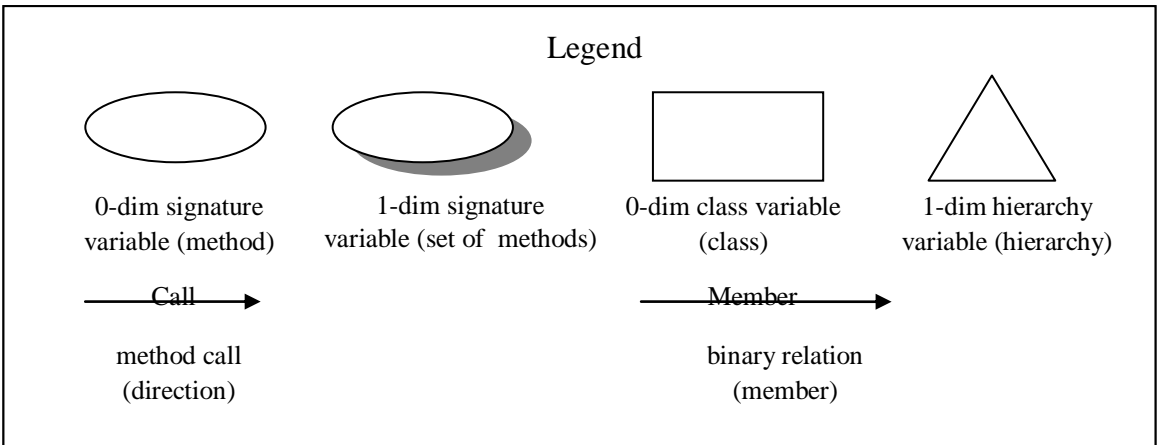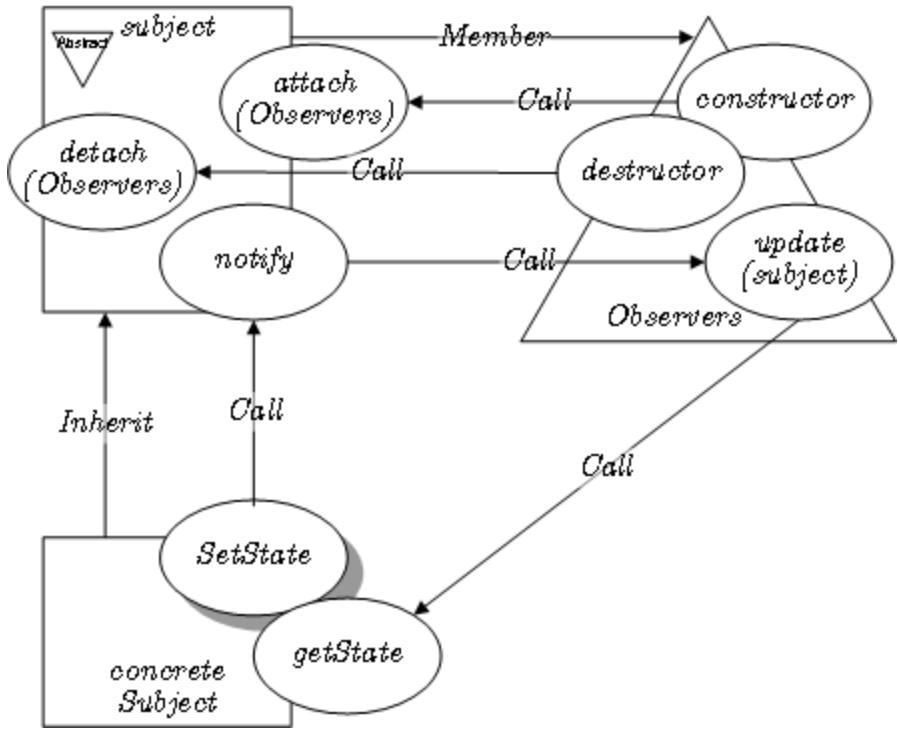
Figure 2.6 Codechart representation of the observer pattern [Eden et al. 2007]

┌─ Observer ───────────────────────────────────────────────────────
│ subject, concreteSubject: CLASS
│ Observers: HIERARCHY
│ getState, notify, attach(Observers), detach(Observers): SIGNATURE
│ constructor, destructor, update(subject): SIGNATURE
│ SetState: PSIGNATURE
│ ──────────────────────────────────────────────────────────────────

45

Abstract(subject)
Inherit(concreteSubject,subject)
Total(Member,subject,Observers)
Total(Call,SetState⊗concreteSubject,notify⊗subject)
Total(Call,notify⊗subject,update(subject)⊗Observers)
Total(Call,update(subject)⊗Observers,getState⊗concreteSubject)
Total(Call,destructor⊗Observers,detach(Observers)⊗subject)
Total(Call,constructor⊗Observers,attach(Observers)⊗subject)

Figure 2.7 Class-Z Schema of the Observer Pattern [Eden et al. 2007]

Figure 2.7 provides the class-z schema of the observer pattern. Following paragraphs explain the symbols used in the schema.

There are two kinds of formulas: *ground* and *predicate formulas*. *Ground formulas* specify the constraints on the entities (i.e., classes and objects) whereas *predicate formulas* specify the relations among sets of entities.

Entities are the elements of the specification language LePUS3 and Class-Z which are class, methods, and method signatures. Each entity has a *type* and a *dimension*. There are seven kinds of *types* in the specification of LePUS3 and Class-Z: *CLASS, PCLASS, SIGNATURE, PSIGNATURE, METHOD, PMETHOD, and HIERARCHY. Type* is used to denote whether an entity is a class (*CLASS)* or a set of classes (*PCLASS*), a method signature (*SIGNATURE*) or a set of method signatures (*PSIGNATURE*), a method (*METHOD*) or a set of methods (*PMETHOD),* or a hierarchy (*HIERARCHY). Dimension* is used to specify whether an entity is an atomic unit or a finite set of atomic units. There are two kinds of dimensions: *dimension-0* or *dimension-1*. An entity of *dimension-0* represents an atomic entity such as a class, a method, or a method signature. An entity of *dimension-1* represents a non-empty, finite set of atomic entities, i.e., a non-empty set of

46

entities of dimension-0. For example, if *Class A* in a Java program, is an entity of dimension-0, then *Class B, which extents A and implements C* in a java program is an entity of dimension-1. A hierarchy of classes is also an entity of dimension-1. The difference between a set of a finite number of classes and a hierarchy of classes is that a hierarchy is also a set of a finite number of classes which has one *root* and all other classes in the set inherit from the root class. So, a *HIERARCHY* is a subset of classes (*PCLASS*) (i.e., an entity of dimension-1).

Entities are of two kinds: constant or variable. Constant entities represent entities that are bound to a specific implementation and variable entities represent entities that are not tied to any specific implementation. For example, as stated by Eden et al., "design patterns and generic elements of application frameworks are not tied in to a particular implementation, their specification therefore requires *variables* rather than *constants"* [Eden et al. 2007]. Constant entities represent specific entities and variable entities range over constant entities.

Relations specify relationships such as *inherits and instance of* that exists among entities. Relations are also of two types: *unary* and *binary*. A unary relation represents an entity of dimension-0. For example, the unary relation of a method is but the method itself, i.e., the unary relation of a method is also known as a *method of dimension-0.* A binary relation represents relationship between two entities of dimension-0. For example, the binary relation *inherits* represents the inheritance relationship between two classes of dimension-0.

According to Eden et al., "declaration is a comma-separated list of constants and variables, TYPE is a type symbol, and formula is a well-formed formula in Class-Z" [Eden et al. 2007].

Predicates or predicate symbols are used to impose the constraints on relations that exist between entities. There are three kinds of predicates symbols that are used in LePUS3 and Class-Z: *ALL*, *TOTAL*, and *ISOMORPHIC*.

Predicate symbol *ALL* is of the following form: *ALL*(*UnaryRelation*,T). ALL specifies an onto relation between UnaryRelation and the entities in T. For example, the predicate formula *ALL*(*Abstract*,Operations⊗collection) specifies that all the methods with the signature represented by *Operations* in class *collection* are abstract [Eden and Gasparis 2009].

Predicate symbol *TOTAL* specifies a total functional relation (*BinaryRelation*) from the entities of one set to the other, i.e., any pair of elements in the set of relations holds the relation. A *TOTAL* predicate is of the form*: TOTAL*(*BinaryRelation*,$T_1$,$T_2$), where $T_1$ and $T_2$ are entities.

Predicate symbol *ISOMORPHIC* specifies the existence of a one-to-one and onto relation from the entities of one set to those of another. For example, *ISOMORPHIC*(*BinaryRelation*,$T_1$,$T_2$) indicates the existence of a subset of the *BinaryRelation,* which is one-to-one and onto, from the set of concrete entities $T_1$ to the set of entities $T_2$.

2.6 Comparison of DisCo/BPSL/LePUS and Need for Another Specification Scheme

DisCO was mainly aimed at formalizing the behavioral aspects of design patterns, hence its characterization of the structural aspects of patterns is not good compared to the behavioral specification aspects. Actions in DisCo are separated from the objects, thus clearly violating the principles of object-oriented design [Hallstorm 2004]. DisCo specifications concentrate mostly on the behavioral aspects, thus largely leaving the structural aspects to be considered by the designer. So, when a designer provides an implementation that satisfies the temporal properties characterized by a particular specification, the designer is responsible for the most part to make sure that the structural aspects do not violate any temporal properties.

As its name suggests, BPSL (Balanced Pattern Specification Language) provides a mostly balanced specification of the structural and behavioral aspects of design patterns, but fails to concentrate on the understandability of the resulting specifications that is critical for the usability of the patterns [Kim and Carrington 2004]. Pattern specification schemes/languages should be understandable and not complex to the users of patterns [Kim and Carrington 2004]. The mathematical notations used for formal specification should not be a burden for software developers. The developers' having to delve into the details of the mathematical foundations of a formalism, before they can understand the meaning of a pattern, constitutes extra and arguably unnecessary work for pattern users.

LePUS3 is still in its initial evolutionary phases. This two-tier programming tool support for the specification, verification, and visualization of design patterns as well as software systems is still a work in progress [Eden and Gasparis 2009]. One of the main limitations of LePUS3 and Class-Z is that they do not really capture or represent the

behavioral elements of design patterns [Eden et al. 2007]. LePUS3 specifies only the structural elements of the design patterns. Since the behavioral elements can also be represented using static relations, Eden et al. have focused on capturing the static or structural elements rather than the dynamic or behavioral elements [Eden et al. 2007].

There have been a few research works that have contributed to the evaluation of the existing formal specification methodologies available for patterns. One of a notable works done in this area is by Chinnasamy [Chinnasamy 2000]. This work identified the merits and demerits of some of the existing formal specification languages such as Contracts [Helm et al. 1990], DisCo [Mikkonen 1998], LePUS [Eden 2000], Constraint Diagram [Lauder and Kent 1998], and Category Description Language (CDL) [Klarlund et al. 1996]. This work identified the preferred specification scheme among the five specification schemes motioned above. The evaluation criteria used by Chinnasamy are [Chinnasamy 2000]: formalism, comprehensiveness, versatility, mathematical foundation, precise visual notation, conciseness, specification of structural and behavioral aspects, specification of constraints, scalability, complementing object notations, support for object-orientation, ease of use, multi-level representation, representation of low-level details, and potential for pattern repository management. These evaluation criteria resulted in the conclusion that "LePUS has many merits that makes it an ideal starting place, which can be enhanced to be a comprehensive language for specification of software design patterns" [Chinnasamy 2000]. In fact, the conclusion resulted in the development of the formal specification language eLePUS. The pros and cons as observed and reported by Chinnasamy about LePUS and DisCo, are listed below in Table 2.2.

| | LePUS | DisCo |
|---|---|---|
| Mathematical Foundation | Strong Mathamatical Model (Higher Order Monadic Logic, Predicate Calculus) | Fair Temporal Logic of Actions |
| Precise Visual Notation | Exists | Nil |
| Conciseness | Very good | Fair |
| Specification of Structural Aspects | Very good | Fair |
| Specification of Behavioral Aspects | Fair | Very good |
| Specification of Constraints | Good | Fair |
| Scalability | Good | Very good |
| Complementing Object Notations | Nil | Nil |
| Support for OOP | Good | Good |
| Easy of Use | Good | Fair |
| Multilevel Representation | Nil | Nil |
| Representation of Low-Level Details | Fair | Good |
| Support for Pattern-Repository Management | Good | Poor |

Table 2.2 Evaluation of LePUS and DisCo [Chinnasamy 2000]

Although a lot of research effort has been spent to formalize design patterns, the existing formalization methods have the following drawbacks: either they are good at capturing the behavioral aspect or they are good at capturing the structural aspect, and the schemes that try to capture both the structural as well as behavioral aspects have failed to concentrate on the understandability of the resulting specification. As mentioned previously, pattern specification schemes/languages should be understandable and not complex to the users of patterns [Kim and Carrington 2004]. The schemes that have

attempted to capture both the structural as well as the behavioral aspects of patterns have actually failed to capture both the essential structural and the essential behavioral elements of patterns, which has led to a lot of overlapping patterns [Chinnasamy 2000]. This has lead to a situation where there is neither a standard methodology for representing patterns nor a standard definition for *what a pattern is* [Agerbo and Cornils 1998] [Bayley and Zhu 2008] [Buschmann et al. 2007] [Chinnasamy 2000] [Dong 2002] [Eden 2000] [France et al. 2004] [Kent and Lauder 2004] [Mikkonen 1998] [Soundarajan and Hallstrom 2004] [Taibi and Ling 2003 B]. The lack of a standard methodology or a definition for representing patterns is mainly due to the reason as mentioned by Taibi and Ling [Taibi and Ling 2003 B] that each specification scheme is based on different mathematical formalisms which reflect their specific author's opinion on *how patterns should be formalized.*

The number of design patterns keeps growing [Buschmann et al. 2007], and the consequences of this growth may be that it will become next to impossible to figure out which pattern solves which problem. This proliferation can also seriously negatively impact, if not destroy, the possibility of using patterns as a common vocabulary among software developers. As Agerbo and Cornils stated over a decade ago, "an overdose of design patterns will eliminate two of the three benefits that design patterns offer: it will make it too laborious to find and use the encapsulated experience, and it will make the common vocabulary too large to be easily comprehended" [Agerbo and Cornils 1998].

Therefore, at this point it is necessary to come up with a formal model that can capture both the structural as well as the behavioral aspects of design patterns by keep in mind the understandability of the formalism behind the specification scheme. The

proposed pattern specification schemes should be understandable and not complex, and the formalism behind the formal specification schemes should not be a burden for pattern users, so that the users will not have to delve into the details of the formalism before they can understand the meaning of a pattern. Also, the specification scheme should enhance the eventual possibility of automated processing capabilities for design patterns.

Two-level grammars (TLG) can provide such a formal model. Specifications written in TLG are understandable due to their natural-language-like vocabulary [Bryant et al. 1986] [Bryant and Pan 1992] [Maluszynski 1984] [Edupuganty 1987]. The close correspondence between the TLG specification and the problem description further enhances understandability [Edupuganty 1987]. The primary reason for the use of TLG as a specification scheme in this research work is its natural-language-like vocabulary, understandability, the capability for data and procedural abstractions provided by the two levels of the grammar, and their support for representing object-oriented constructs [Lee 2003] [Edupuganty 1987] [Bryant and Pan 1992]. Formal grammars have been used to generate and formalize programming languages. This formalization has not restricted the usability of programming languages. Therefore, it seems appropriate to provide a formal specification scheme for patterns using grammars and other traditional approaches that have been used for programming languages. Of course, the original van Wijngaarden two-level grammar [van Wijngaarden et al. 1975] may not be directly used for the specification of object-oriented design patterns since van Wijngaarden two-level grammar were not aimed at defining object-oriented units. TLG has been tailored by authors like Lee, Edupuganty, and Bryant [Lee 2003] [Edupuganty 1987] [Bryant et al.

53

1986] to make it suitable for the representation of object-oriented units such as classes, objects, and methods.

Although a lot of research effort has been spent to formalize design patterns, there are several objections to the formalisms used and the formalization from the pattern community.

- Formal specifications contribute little or nothing to understanding when and how to use a pattern. As Buschmann et al. put it [Buschmann et al. 1996], "formalizing the solution makes it harder to grasp the key ideas of the pattern… programmers need concrete information that they can understand, not an impressive formula".

- Patterns are abstractions, or generalizations, and therefore are meant to be vague, ambiguous, and imprecise. If they are specified in a precise form, or expressed in mathematical terms, they are no longer patterns [Buschmann et al. 1996].

- There is no fixed element in patterns and everything can be changed about them. In other words according to Coplien, if "the basic structure is fixed… this isn't patterns anymore" [Coplien 1991].

However, the formal specification of patterns is not going to replace the textual/graphical descriptions, rather it will complement the existing descriptions to achieve well-defined semantics, to allow for rigorous reasoning about them, and to facilitate tool support, and to enhance the understandability of their semantics [Taibi and Ling 2003 B]. The following section provides a detailed description of two-level grammars.

2.7 Two-Level Grammars

Two-level grammars (TLG) were introduced by van Wijngaarden [Wijngaarden 1965] to define the context-free and context-sensitive syntax of ALGOL 68 [Wijngaarden 1975]. A two-level grammar is usually viewed as an extension to a context-free grammar with an infinite number of productions.

The definition of TLG given below is based on two references[DeGraaf and Ollongran 1984] [Maluszynski 1984]. A TLG is defined as a triple W = [(Σ, T), (N, HR, L), (M, MR)], where Σ is a finite alphabet of terminals and T is a finite alphabet called *orthovocabulary*, (N, HR, L) is called the hyper-level and consists of N, a subset of (T ∪ M)*, that is a finite set of hyper-notions, HR, a subset of ((Σ ∪ N)*N(Σ ∪ N)*) x (Σ ∪ N)*, that is a finite set of hyper-rules, and L ∈ N that is the start notion, and (M, MR) is called the meta-level and consists of M, a finite set of meta-nonterminals for which M ∩ T =∅, and MR, a subset of ((T ∪ M)*M (T ∪ M)*) x (T ∪ M)*, that is a finite set of meta-rules. In a two-level grammar, a hyper-notion in the left-hand side of the hyper-rules is surrounded by any number of elements from either the set of terminals or hyper-notions. A meta-nonterminal in the meta-rules is surrounded by any number of elements from either the orthovocabulary or the set of meta-nonterminals. Consider the following TLG example [Saacks and Hassell 1989]:

Σ = {else}, T = {arith, Boolean, designational, exp, simple, if clause}, N = {<X exp>, <simple X exp>, <if clause>}, and M = {X}.

Hyper-rule:

    <X exp> ::= <simple X exp> | <if clause><simple X exp>else<X exp>

Meta-rule:

    X ::= airth | bool | designational

The above TLG will produce for instance " <arith exp> ::= <simple arith exp> | <if clause><simple arith exp>else<arith exp>" since the hyper-rules "<X exp> ::= <simple X exp> | <if clause><simple X exp>else<X exp>" expands into :

<arith exp> ::= <simple arith exp> | <if clause><simple arith exp>else<arith exp>

<bool exp> ::= <simple bool exp> | <if clause><simple bool exp>else<bool exp>

<designational exp> ::= <simple designational exp> | <if clause><simple designational exp>else< designational exp>

A TLG has two sets of production rules: *hyper-rules* and *meta-rules*. Hyper-rules form the prototype for the context-free productions that are used in conjunction with the meta-rules to form the infinite number of productions. In other words, these two sets of rules define "the set of type domains and the set of function definitions operating on those domains" [Caol et al. 2002]. Meta-rules are context-free productions containing two kinds of symbols: *meta-notions* as the non-terminals and *proto-notions* as the terminals. Each meta-rule or meta-production specifies all the production alternatives for a given meta-notion. Meta-notions define the type domains and hyper-rules define the set of function operating on the domains defined by the meta-notions.

It has been proven that the production rules of two-level grammars can be used to simulate a Turing machine [Sintzoff 1967]. Therefore, TLG is capable of providing a complete formal specification of programming languages and systems [Edugupanty 1987]. So, it makes sense to use two-level grammar as a formal specification language for design patterns. Moreover, TLG could be used to define the operational and axiomatic semantics of programming languages and systems [Uzgalisn1973], which means that TLG specifications can be implementable when appropriate interpretation algorithms are

provided. Specifications written in TLG are understandable due to their natural-language-like vocabulary, and the close correspondence between a TLG specification and the corresponding problem description further enhances the understandability of the specification [Edupuganty 1987] [Maluszynski 1984] [Lee 2003] [Bryant et al. 1986].

The descriptive power of two-level grammars enabled their use as a general model for computation [Sintzoff 1967]. TLGs have been used as a tool for defining programming languages [Maluszynski 1984] and one of the software reusable units called *programming scheme* [Saacks and Hassell 1989]. Significant work by Sintzoff [Sintzoff 1967] and van Wijngaarden [Wijngaarden 1975] demonstrated that production rules of a two-level grammar can be used to simulate a Turing machine, which resulted in using TLGs for the specification and generation of programming languages such as A Simple Programming Language (ASPLE) [Cleavland and Uzgalis 1977] and Subject Language (SL) [Edupuganty 1987]. Two-level grammars have been used to define the syntax and both the static and dynamic semantics of programming languages [Cleavland and Uzgalis 1977]. TLGs have also been used as implementable meta-languages for implementing the axiomatic, operational, and denotational semantics of programming languages [Edupuganty 1987]. More recently, two-level grammars have been used as an executable formal specification language for programming languages [Bryant et al. 1986], database applications [Furtado et al. 1983], knowledge-base systems, and general software systems [Bryant and Pan 1992].

Formal definitions of TLG found in the literature have small variations, although all of them maintain the same basic structure [Edupuganty 1987]. For example, in formalizing programming schemes, a restriction is imposed on the hyper-rule and the

restricted two-level grammar is used to project the representation of programming schemes [Saacks and Hassell 1989]. The definition of a restricted two-level grammar as given by Saacks and Hassell is: a triple $W = [(\Sigma, T), (N, HR, L), (M, MR)]$, where $\Sigma$ is a finite alphabet of terminals and T is a finite alphabet called *orthovocabulary,* (N, HR, L) is called the hyper-level and consists of N, a subset of $(T \cup M)^*$, that is a finite set of hyper-notions, HR, a subset of $(N(\Sigma \cup N)^+) \times (\Sigma \cup N)^*$, that is a finite set of hyper-rules, and $L \in N$ is the start notion, and (M, MR) is called the meta-level and consists of M, a finite set of meta-nonterminals for which $M \cap T = \phi$ , and MR, a subset of M X T is a finite set of meta-rules [Saacks and Hassell 1989]. So, the left-hand side of the hyper-rule of a restricted two-level grammar always has a hyper-notion first and then at least one element from either the terminals or from the hyper-notions. The left-hand side of the meta-rules will only have meta-nonterminals, and the right-hand side only hyper-notions. The hyper-rules and the meta-rules of the general two-level grammar are less restrictive [Saacks and Hassell 1989]. As already mentioned earlier (see the TLG definition given in Section 2.7), in a conventional two-level grammar, a hyper-notion in the left-hand side of the hyper-rules is surrounded by any number of elements from either the set of terminals or hyper-notions. A meta-nonterminal in the meta-rules is surrounded by any number of elements from either the orthovocabulary or from the set of meta-nonterminals.

In addition to the changes proposed on the formal properties of TLGs, there are some changes imposed on the structure of the two-level productions as well. For example, Saacks and Hassel imposed some changes to the structure of the TLG production rules and called the resulting grammar as the *Restricted Two-Level Grammar (RTLG)* to formalize *programming schemes* [Saacks and Hassel 1989]. Edupuganty made

some changes to the structure of the TLG production rules to use TLG as an implementable metalanguage for axiomatic semantic of the programming language called as *Subject Language* [Edupuganty 1987]. Example 1 below provides an example of a RTLG and Example 2 provides an example of the TLG.

Example 1: Consider a search scheme with binary and linear search approaches. The hyper-rule of the TLG for formalizing the first level of this search scheme, as given by Saacks and Hassel [Saacks and Hassel 1989], is as follows:

**HR:** SP<search> ::= SP<init_search> while SP<succful_cond> AND

SP<unsuccful_cond> do SP<get_next> endwhile if

SP<succ> then SP<found> else SP<not_found> endif

The meta-rule for this level is: **MR:** SP ::= <linear> I <binary>

Example 2: In the formal specification of the semantics of a programming language called the Subject Langugae (SL), Edugupany specified hyper-rules as functions and meta-rules as arguments to those functions [Edupuganty 1987]. What follows is the set of hyper-rules and meta-rules that specify the declaration of a programming block.

Program ID DECLARATIONS SEMICOLON begin

CONCRETE_STMTS end with input FILE1:

Synthesize environment ENV1 from DECLARATIONS with initial env EMPTY

Check static semantics of CONCRETE_STMTS giving ABSTRACT_STMTS

given end ENV1,

Allocate storage for variables in env ENV1 giving STORE1 and env ENV2,

Execution of ABSTRACT_STMTS transforms state env ENV2 store STORE1

Input FILE1 output EMPTY into state env ENV2 store STORE2

Input FILE2 output FILE3

Result of interpretation is FILE3.

The starting hyper-rule of a programming block declaration is: *Program ID DECLARATIONS SEMICOLON begin* where capitalized words are the meta-rules and the lower case letters are the proto-notions. The starting rule starts with the word *Program* then it assigns an identifier *(ID)* to the program, then there is a meta-rule to specify the declarations, then the last meta-rule SEMICOLON is the semicolon symbol (;), and the last word of the starting hyper-rules is *begin* which marks the starting line of the program body. Line 2 (CONCRETE_STMTS end with input FILE1) is a hyper-hyper-notion which has three hyper-alternatives:

1. Synthesize environment ENV1 from DECLARATIONS with initial env EMPTY. Check static semantics of CONCRETE_STMTS giving ABSTRACT_STMTS given end ENV1

2. Allocate storage for variables in env ENV1 giving STORE1 and env ENV2

3. Execution of ABSTRACT_STMTS transforms state env ENV2 store STORE1 Input FILE1 output EMPTY into state env ENV2 store STORE2 Input FILE2 output FILE3

The first hyper-alternative synthesizes the variables in environment (ENV1) which are in the declarations (DECLARATIONS) and checks the static semantics of the concrete statements in the program body. The second hyper-alternative allocates storage for the variables in STORE1, and the third hyper-alternative executes the statements and

stores the results in FILE3. A number of the meta-rules that can be used in the hyper-rules are listed below.

PROGRAM :: program ID SEMICOLON BLOCK

BLOCK :: DECLARATIONS begin CONCRETE_STMTS end

CONCRETE_STMTS :: CONCRETE_STMT SEMICOLON

CONCRETE_STMTS; CONCRETE_STMTS

These structural differences to TLGs occur when the domain of applications of the two-level grammar changes. In Example 1, the domain of the two-level grammar is programming schemes. Therefore, the meta-notions and the hyper-rules reflect the structure of the programming schemes. In Example 2, the domain of the two-level grammars is block-structured programs, therefore the structure of the TLG reflects the structure of the block-structured program. This close correspondence between the TLG specification and the structure of the application domain that it describes, enhances the understandability of the problem [Edupuganty 1987].

Since the formalization of object-oriented design patterns requires the formalism to provide support for object-oriented constructs, the domain of the TLG will have to be of objects and thus the TLG can be defined in the context of classes [Lee 2003].

2.7.1 Formal Specification of Reusable Units Using Two-Level Grammars

TLGs have been used for the formal specification of reusable software units called *programming schemes* [Saacks and Hassell 1989]. According to Saacks and Hassell, a programming scheme is a problem solving approach that contains "only the essential features of the process that are needed to solve the problem". A programming

scheme is an abstraction of a problem solving scenario in programming. Initially, programming schemes were described in natural languages. It soon became apparent that ambiguous, imprecise, and indefinite description of programming schemes needed a more formal approach. Saacks and Hassell [Saacks and Hassell 1989] worked on formalizing programming schemes utilized two-level grammars toward the formal specification of the context-free and context-sensitive syntax of programming schemes. The semantics of programming schemes were defined using the denotational semantics approach.

Two-level grammars were used for defining the syntax of programming schemes because TLGs can accurately capture and represent the hierarchies inherent in programming schemes [Saacks and Hassell 1989]. The hyper-rules represent the common rules applicable for a specific programming scheme and the meta-rules represent various decision options that are essential in generating the *scheme representation*. A *scheme representation* is the concrete machine-dependent representation of an abstract programming scheme [Saacks and Hassell 1989]. The elegance of two-level grammar is that the hyper-rules can be used to capture and represent the basic structure of the abstract unit being formalized, and the meta-rules can be used to provide the design choices and decisions that are essential for the concrete realization of the abstract units by passing to them as arguments [Edupuganty 1987] [Saacks and Hassell 1989]. Hyper-rules and meta-rules together can be used to enforce content-dependent conditions. Meta-rules can also be used to specify data-types and variables [Edupuganty 1987]. This capability of data and procedural abstractions provided by the TLG makes it suitable as a formal specification language for programming languages, general software systems, and reusable units.

2.7.2   Formal Specification Language for Design Patterns

The concept of using the two-level grammar notation as a formal specification language is not new. TLGs have been used as a formal specification language for general software systems [Bryant et al. 1986]. TLG specifications use the generative approach to automatically generate a software system from its TLG specification. This research work focused on refining this concept into using the TLG as a formal specification language for design abstractions. TLG specifications of languages and general software systems are actually descriptions of recursive functions, and a TLG representation of a system can be derived by using a recursive definition of a given problem [Edupuganty 1987].

TLGs have been used as a specification language for object-oriented software systems [Lee 2003], where the hyper-rules of the TLG specifications define the functions that operate on the object-oriented domain. Lee defines the grammars in the context of a class where the meta-notions define the instance variables of the class and the hyper-rules define the methods that take part in the classes [Lee 2003].

Since TLGs are used as a specification language for representing object-oriented design patterns in this thesis work, this work deal with the TLGs whose hyper-rules define functions that will act on the object-oriented constructs. Therefore, the TLGs are represented in the context of a class, in other words, the hyper-rules of the TLGs act as the methods of a class, and the meta-notions and the proto-notions act as instance variables of a class. A detailed representation of the TLG in the context of a class is described in Chapter III.

# CHAPTER III

# TLG IN THE CONTEXT OF A CLASS

This research work is in the same scope of some of the existing formal specification approaches of design patterns such as eLePUS, LePUS3 and Class-Z, DisCo, and BPSL (for a description of these specification schemes, see chapter II). These formal specification languages are based on object-oriented languages and have focused on a subset of the GoF design patterns [Eden et al. 2007] [Mikkonen 1998] [Taibi 2007]. The formal specification of this research work focuses on the solution element of a pattern (for a list of all the essential elements that are used to specify or represent a pattern, see Chapter II). The solution element corresponds to the structure, participants and collaborations sections of the GoF pattern form.

Based on the related literature discussed in Chapter II, this chapter further describes the use of TLG as a specification language for the object-oriented design patterns. After presenting the TLG formal specification language in this chapter, the next chapter demonstrates the approach in the context of an example. The observer design pattern is used to demonstrate the use of TLG as a formal specification language.

3.1 Introduction

Before starting the use of TLG as a formal specification language for the object-oriented design patterns, the section elaborates on how the rules and notions of TLG should be read and written. As mentioned before (section 2.7), TLG has two sets of production rules: hyper-rules and meta-rules. Three kinds of notions are used in these two sets of production rules: hyper-notions, meta-notions, and proto-notions. Hyper-rules can contain meta-notions and proto-notions. The following section briefly explains about how to read and write rules and notions in TLG. These explanations are mostly based on the work of Cleaveland and Uzgalis [Cleaveland and Uzgalis 1977].

A **hyper-rule** is the first set of production rules in TLG that may contain proto-notions and meta-notions. Hyper-rules should be written in the following way.

- A colon separates the left-hand and right-hand side of a hyper-rule.

- A comma is used to represent the set of symbols that can be included in the production rules.

- A semicolon is used to indicate the alternative production rules for the right-hand side of a rule.

- A period is used to indicate the end of a hyper-rule.

A *meta-rule* is a production rule that defines a single meta-notion. In a meta-rule, the left-hand side is the meta-notion that needs to be defined and the right-hand side can have a sequence of proto-notions and meta-notions. A *meta-notion* is denoted by a string of upper and lower case characters, and the meta-rules are written using the following rules:

- A double colon (**::**) is used to separate the left- and right-hand sides of a meta-rule.

- A semicolon is used to indicate alternative production-rules for the right hand side.

- A period is used to indicate the end of a meta-rule.

- Meta-notions may appear anywhere in a hyper-rule, i.e., they can appear either on the right-hand side or the left-hand side of a hyper-rule.

For example, consider the following hyper- and meta-rules.

Hyper-rule:

NOTION list: NOTION; NOTION, NOTION list.

Meta-rule:

NOTION :: identifier; digit; letter; numeral.

A *proto-notion* is a sequence of lower case letters. Boldface characters are used to denote the non-terminals that correspond to a terminal symbol in the target language.

**block** stands for nonterminal <block>.

**begin symbol** stands for keyword **begin**.

3.2 TLG and Design Patterns

In order to represent object-oriented design patterns, TLG was modified by including in the vocabulary items such as class, inheritance, and member to represent the object-oriented building blocks and the relationships among classes, and functions [Lee 2003] [Liu et al. 2005] [Edupuganty 1987] [[Bryant et al. 1986]. Lee used TLG as a formal specification language to represent the requirements of an object-oriented software system [Lee 2003]. Liu and his colleagues used TLG++ to represent object-oriented software units [Liu et al. 2005]. TLG++ is an object-oriented extension of TLG

[Zhao 2006]. In these methods, the hyper-rules define the set of functions and the meta-rules define the set of type domain in the domain of objects. Function definitions act on the type domains to produce the target language. The target language in the research work presented in this thesis report is design patterns. So, the functions and type-domains are defined in the context of object-oriented units. This is based on the fact that design patterns are recurring themes in object-oriented software systems. Therefore, TLG in this thesis work is defined in the context of a class, where hyper-rules define the functions in a class and the type domains define the instance variables of the class.

The following subsections introduce the essential concepts and basic building-blocks of object-oriented design that can be used to construct design patterns. They also outline how these building blocks can be represented in TLG.

3.2.1 Building Blocks of Object-Oriented Design Patterns

The basic building blocks of object-oriented design patterns are the *class*es, *objects, operations, attributes, and relations between classes such as inherit* and *implements*. A class defines the available characteristics and behavior of a set of similar objects. A class is an abstract definition. It is made concrete at run-time when objects based upon the class are instantiated and take on the class' behavior. In the TLG representation of design patterns, both classes and operations are primitive elements. The predominant participants of all the GoF design pattern are *classes, methods, and objects* [Eden 2002]. Classes and methods are static entities and object is a run time entity. This research work focused only on capturing static aspects.

3.2.2 Static vs. Dynamic

Object-oriented design patterns have two kinds of properties: structural and behavioral. Structural aspects are the static elements of design patterns that can be dealt with at compile time. For example, classes, interfaces, attributes, operations, and association are structural elements of a design pattern. Behavioral aspects of design patterns are elements of patterns that can only be dealt with during run time. For example, run time events and message passing among classes and objects can be considered as behavioral aspects. This research work focused on capturing the static aspects of design patterns because static aspects serve as the foundation to build the dynamic properties. Capturing the dynamic aspects of design patterns is beyond the scope of this thesis and it is part of the future work. Since this research works' focus is on capturing the static aspects, the following section discusses the representation of classes and methods in TLG.

3.2.3 Classes

To represent object-oriented units, a TLG definition can be structured into a class. The syntax of a TLG class is:

**Class** Identifier [extends Identifier-1, Identifier-2, Identifier-3, . . . , Identifier-n]

{instance variable and method declaration}

**End Class** [Identifier-1]

Identifier designates the name of the class. Identifier-1 through Identifier-n specify the names of the classes from which the current class inherits. In the above class structure, square brackets are used to indicate the optional specification part. Identifier-1

can inherit from Identifier-2 to Identifier-n. So, the *extends* clause is optional. The instance variables defined in a class are the meta-rules, and the methods are the hyper-rules. The TLG class is not a class as defined in an object-oriented programming language, rather it just an abstract representation of a participant in patterns that can be eventually implemented as a class. The TLG class declaration encapsulates the meta-rules and the hyper-rules. The Meta-rules specify the instanced variables and objects in the class, and the hyper-rules define the functions that would operate on the meta-rules.

There are situations in which a class should declare the structure of an abstraction without providing a complete definition for all or some of the hyper-notions, also all the classes that inherit from this base structure should provide the details for each hyper-notion. Such an abstract structure is an abstract class that determines the nature of the hyper-rules that the inheriting classes must implement. The difference between a class and an abstract class is that a class is a structure in which all the meta-rules and hyper-rules are defined, whereas in an abstract class, some of the hyper-notions can be left undefined. In other words, the right-hand side of some hyper-notions can be left undefined. All hyper-notions whose right-hand side is empty will be overwritten by the classes that inherit from the abstract class. If any of the hyper-notion specified by a hyper-rule in the abstract class is empty, then it is an abstract hyper-rule, and the inheriting classes should provide a definition for that hyper-notion.

**abstract Class** abstract-class-name-1

**abstract** abstract-hyper-notion: .

hyper-notion: hyper-alternative-1; hyper-alternative-2; . . . ; hyper-alternative-n.

End **Class**

**Class** derived-class-name **extends** abstract- class-name-1

abstract-hyper-notion: hyper-alternative-1; hyper-alternative-2; . . . ;

hyper-alternative-n.

// other hyper-notions and meta-notions

End **Class**

Hyper-rules are declared abstract when they are required to be present in the classes that inherits the abstract class. The inheriting classes can provide their own definition for the hyper-notions. To declare a hyper-notion as abstract, the following structure is used:

**abstract** abstract-hyper-notion: .

The right-hand side of an abstract notion is empty and the overriding classes provide the necessary hyper-alternatives to this hyper-notion. Any structure that contains one or more abstract methods will also be declared abstract. These abstract classes can be assigned any object of the same type, i.e., the abstract class objects can be assigned an object of a class that inherited from this abstract class. As an example, consider the TLG specification of the observer pattern given below. It has one abstract class and an interface definition followed by the implementations of the abstract class and the interface. The meta-notion of the class *concreteObserver* assigns the object of any concrete subject class to the object of the abstract class *Subject.*

**Class** abstract Subject
//meta-notions
List observers_list :: (concreteObserver)*.

//hyper-rules
Attach: observers_list.**Add**.

Detach: observers_list.**Remove**.

Notify: observers_list.**Update**.

**End Class**

**Class** concreteSubject **extends** Subject

//hyper-rules

setState: this.Notify.

getState: this.currentState.

**End Class**

**Interface** Observer

//hyper-rules

Constructor: .

Destructor: .

Update: .

**End Interface**

**Class** concreteObserver **implements** Observer

//meta-notions

Subject subObject:: (concreteSubject)$^+$.

//hyper-rules

Constructor: subObject.Attach.

Destructor: subObject.Detach.

Update: subObject.getState.

**End Class**

A class in TLG can inherit from another TLG class when additional methods need to be included. A class can inherit from any number of parent classes. Inheritance allows a class to extend another class. Two kinds of inheritance can be represented using TLG: *class inheritance* and *interface inheritance*. In class inheritance, a new class extends from another class, i.e., there is a base class and the new class inherits the hyper-rules and meta-rules of the base class. In interface inheritance, a new class implements the hyper-rules defined as part of the interface. Class inheritance is represented as follows:

**Class** base-class-idetifier-1

//meta-rules and hyper-rules

End **Class**

71

**Class** Identifier-1 **extends** base-class-idetifier-1

    {instance variable and method declaration}

End **Class**

Class inheritance is represented using the keyword *extends*. The identifier following the keyword *extends* is the name of the base class(es) from which class Identifier-1 inherits. If a class inherits form more than one class, the base classes are listed separated by commas.

**Class** Identifier-1 **extends** base-class-idetifier-1, base-class-idetifier-2, . . . , base-

    class-identifier-n

    {instance variable and method declaration}

End **Class**

Interface inheritance is represented using the keyword *implements*. An interface is a specification for a set of hyper-rules that a class, which implements the interface, must conform to. In other words, an interface fully abstracts a class specification from its implementation. The class that implements the interface must provide an implementation for each hyper-notion specified in the interface. So, by making a class an *interface*, the hyper-rules can specify what has to be done and not how it has to be done.

Sometimes in a pattern, all the objects that are created during run time should conform to the same interface. Consider the *Abstract Factory* pattern discussed by Gamma et al. [Gamma et al. 1995]. The intend of the *Abstract Factory* pattern is that to provide "an interface for creating families of related or dependent objects without specifying concrete classes" [Gamma et al. 1995]. For example, let's examine the following situation as discussed by Gamma et al. [Gamma et al. 1995]:

72

Consider a user interface toolkit that supports multiple look-and-feel standards such as Motif and Presentation Manager (PM). Different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons. 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 an application make it hard to change the look and feel later. We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There is also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. The 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.

The WidgetFactory has the following participants:

**AbstractFactory (WidgetFactory):** 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):** defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
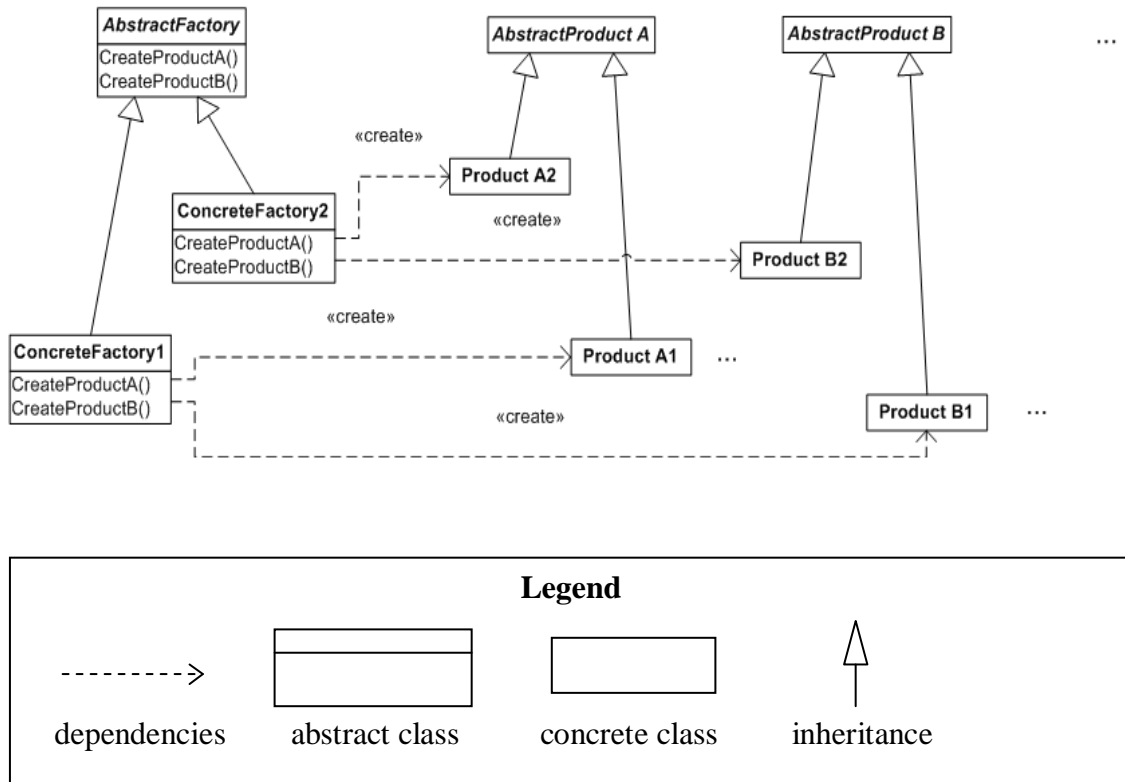
Figure 3.1: Structure of Abstarct Factory Pattern [Gamma et al. 1995]

By defining an interface, the abstract factory pattern declares the specification that should be present for creating each basic kind of class that wants to use the interface. So, when clients call these classes to obtain the functionality, the clients can get the needed functionality conforming to the specifications of the interface [Gamma et al. 1995]. This point can be clarified further by understanding the following representation of the Abstract Factory pattern.
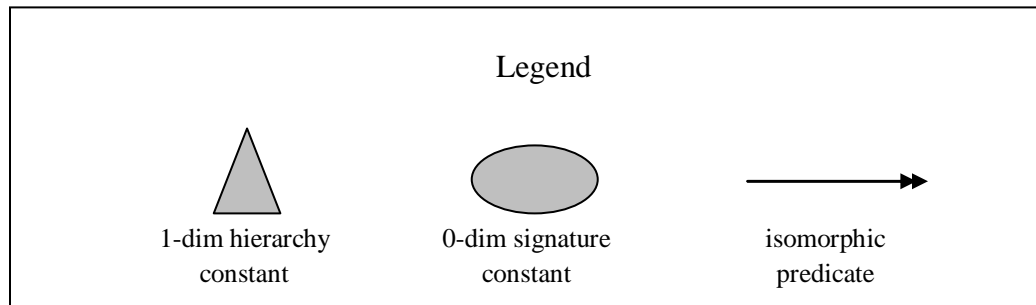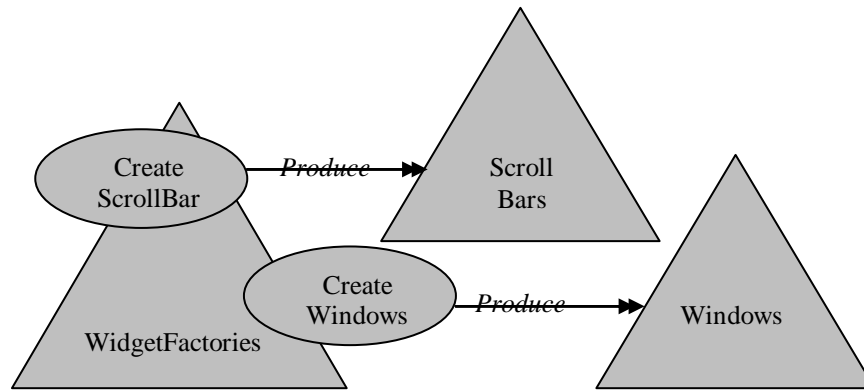
Figure 3.2: Representation of the Widget Factory in LePUS3 [Eden et al. 2007]

There is an interface that declares the specifications of the widgets, which is the *WidgetFactories* interface and there is an interface for each kind of widget such as *ScrollBars, Windows, etc.* that extend from the *WidgetFactories* interface. When an application needs an instance of a scrollbar, the particular concrete scrollbar instance overrides the methods (such as *dimensions* and *alignment (horizontal or vertical))* in both interfaces *WidgetFactories* and *ScrollBars* to create the new instance of a scrollbar. So, the hyper-rules and the meta-rules can be defined in a very general way with the guarantee that, by only using the hyper-rules defined in the interface, all the classes which implement that interface will have defined implementations for all the hyper-rules.

In a TLG specification, interfaces are syntactically similar to classes but they lack meta-notions and the right-hand side of the hyper-rules will be empty. So, interfaces specify the hyper-notions without making assumptions about how the hyper-notions will be defined by the classes that will use this interface. Once an interface is defined, any number of classes can use it and any number of other interfaces can inherit from it. To use an interface, a class must provide definition for the set of hyper-notions specified in the interface. However, each class can provide its own definition for each of the hyper-notions. Structure of an interface is as follows:

**Interface** interface-name-1

       meta-notion:: .

       hyper-notion::.

End **Interface**

To implement an interface, a class should include the *implements* clause in its definition, and then provide the definition for each hyper-notion defined by the interface. The identifiers following the keyword *implements* are the names of the interfaces from which class Identifier-1 inherits.

The general form of a class that includes the implements clause follows:

**Class** Identifier-1 **implements** interface-name-1

       {instance variable and method declaration}

End **Class**

If a class implements more than one interface, the interfaces are separated with commas.

**Class** Identifier-1 **implements** interface-name-1, interface-name-2, . . . ,

interface-name-n

      {instance variable and method declaration}

End **Class**

Interfaces can be extended. When an interface is extended, the implementing class of that inherited interface should provide definition for the hyper-notions defined in the base interfaces and the inheriting interface. For example, consider the specification of two interfaces *A* and *B* and a class D. Class D provides the definitions for the hyper-notions *A*, *B*, and *C* that are declared in the Interfaces *A* and *B*.

**Interface** A

      A:.

      C:.

End **Interface**

**Interface** B **extends** A

      B:.

End **Interface**

**Class** D **implements** B

      A: hyper-alternative-A1; hyper-alternative-A2; . . . ; hyper-alternative-An.

      B: hyper-alternative-B1; hyper-alternative-B2; . . . ; hyper-alternative-Bm.

      C: hyper-alternative-C1; hyper-alternative-C2; . . . ; hyper-alternative-Cl.

End **Class**

3.2.4 Functions

As mentioned earlier (see Section 3.1), the TLG definition of a class consists of hyper-rules and meta-rules. Hyper-rules define the function that operates on the meta-rules to produce the target language, i.e., patterns. The general form of hyper-rules is:

Hyper-notion : hyper-alternative-1; hyper-alternative-2; . . . ; hyper-alternative-n.

The hyper-alternatives specify the alternative rules that can be chosen when a hyper-notion or a function is invoked. Hyper-alternatives have the same format as the hyper-notions. If each hyper-alternative consists of multiple rules, they are separated using commas as shown below:

hyper-notion-1 : hyper-alternative-11, hyper-alternative-12, . . . , hyper-alternative-1i;

hyper-alternative-21, hyper-alternative-22, . . . , hyper-alternative-2j;

hyper-alternative-31, hyper-alternative-32, . . . , hyper-alternative-3k.

So, when hyper-notion-1 is invoked, it can either choose *hyper-alternative-11, hyper-alternative-12, . . . , hyper-alternative-1i* or *hyper-alternative-21, hyper-alternative-22, . . . , hyper-alternative-2j* or *hyper-alternative-31, hyper-alternative-32, . . . , hyper-alternative-3k.*

In object-oriented languages, functions are defined inside a class. Functions can have no parameters or they can take one or more parameters, and the parameters are variables that take the value of the arguments passed to functions when they are called. When a function is defined in the form of a TLG, a parameter list is not provided to it because meta-notions act as parameters to the functions. For example,

T :: **int**; **string**; **Boolean**.

T Expression: simple T expression; **if** <simple T expression> **else** <simple T expression>

Where *T Expression* is a hyper-rule and there are two hyper-alternatives given for the hyper-rules: one is the *simple T expression* and the other alternative is *if <simple T expression> else <simple T expression>*. Based on the value of *T*, the hyper-rule will be generated. If the meta-notion *T* is *int*, the hyper-rule will be:

int Expression : simple int expression; if <simple int expression> else <simple int expression>.

Therefore, meta-notions act as a parameter list for the hyper-rules.

A hyper-notion in a class is represented using the class name in which the method is present, followed by a 'dot' operator which in turn is followed by the name of the hyper-notion. Consider the following example with two classes: *class-name-1* and *class-name-2*.

**class** class-name-1

       hyper-notion-1:hyper-alternative-1.

**end class**

**class** class-name-2

       hyper-notion-2: class-name-1.hyper-notion-1.

**end class**

The class *class-name-1* has a hyper-notion *hyper-notion-1*. To access *hyper-notion-1* in the class *class-name-2*, the representation *class-name-1. hyper-notion-1* is used.

Since hyper-rules are representing methods in the design patterns, they are considered to have return types, including void and other return values, to the assigning routines. Consider the following example:

**Class abstract** Subject

    //meta-notions
        List observers_list:: (concreteObserver)[*].

    //hyper-rules
        Attach: observers_list.**Add**.
        Detach: observers_list.**Remove**.
        Notify: observers_list.**Update**.

**End Class**


**Class** concreteSubject **extends** Subject

    //hyper-rules
    setState: this.Notify.
    getState: this.currentState.

**End Class**


Here, the values returned by the functions Add, Remove, and Update in the abstract class Subject are void. The currentState function (in the right-hand side of the getState function) in concreteSubject will return the current state of the object of the class concreteSubject.

Hyper-notions can also be defined based on a type. For example,

    Subject concreteSubject : concreteSubejct-1.

Here the hyper-notion *concreteSubject* is of type Subject and it can be assigned any objects of the type *Subject*.

3.2.5 Types

When meta-notions are defined in a TLG class, strong typing of the meta-notions are achieved by assigning a types to each meta-notion. These meta-notions can be identifiers or collections such as lists and has-tables. The types can be primitive data types, i.e., int, string, char, and Boolean. Structure of a meta-notion is given below.

meta-notion :: meta-notion-1; meta-notion-2; . . . ; meta-notion-n; proto-notion-1;

proto-notion-1; . . . ; proto-notion-n.

Meta-notions can also be called domain identifier. Domain identifiers are used in conjunction with the functions (i.e., hyper-rules) to produce the target language. The right-hand side of the meta-notions can contain a combination of meta-notions and proto-notions. Proto-notions are terminal symbols of TLG, they are represented using lower case letters.

Type :: int; char, string; Boolean.

int :: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.

char :: a; b; c; d; . . . ; z.

Boolean :: true; false.

string : char, char.

The right-hand side of the hyper-rules above, int, char, and Boolean are proto-notions.

Domain identifiers can be assigned a type using the following format:

Type meta-notion :: meta-notion-1; meta-notion-2; . . . ; meta-notion-n; proto-notion-1; proto-notion-2; . . . ; proto-notion-n.

Type :: int; char, string; Boolean.

int :: 0; 1; 2; 3; 4; 5; 6; 7; 8; 9.

char :: a; b; c; d; . . . ; z.

Boolean :: true; false.

string : char, char.

When assigning a meta-notion to another meta-notion, it is important to keep in mind that the type of hyper-alternative assigned to the hyper-notion must be compatible with the hyper-notion. For example, consider the following class.

**Class** concreteObserver **implements** Observer
  //meta-notions
  Subject subObject :: (concreteSubject)$^+$
//hyper-rules
  Constructor: subObject.**Attach**.
  Destructor: subObject.**Detach**.
  Update: subObject.**getState**.
**End Class**

The meta-notion subObject can be assigned one or more concreteSubject objects if and only if the concreteSubject objects *extends* the abstract class Subject.

A domain identifier can also be a type of collection. A *collection* is the same as the concept of a *set* in mathematics. If a meta-notion can be assigned more than one abstract object, then the meta-notion should be configured as a sequence of abstract data structure that is simply an ordered collection of values. So, this abstract data structure can be treated as a special case. These collection variables can be a *List* or a *Set.* Collection interfaces are considered to have predefined hyper-rules that provide basic operations such as *adding new elements to the collection, removing elements from the collection,* and *updating elements in the collection.* These functions are considered to be functions in the target language and as a result they will be written as bold characters. For example,

**Class abstract** Subject

//meta-notions
List observers_list:: (concreteObserver)$^*$.

//hyper-rules
Attach: observers_list.**Add**.
Detach: observers_list.**Remove**.
Notify: observers_list.**Update**.

**End Class**

# CHAPTER IV

# TLG SPECIFICATION OF THE OBSERVER DESIGN PATTERN

## 4.1 Introduction

To illustrate the TLG specification approach, TLG specification of the observer pattern, which was given in Chapter II, is given below. The TLG specification of the observer pattern was derived based on its natural language specification as given by Gamma et al. [Gamma et al. 1995].

## 4.2 Observer Pattern

The TLG specification of the observer design pattern is split into four subsections to represent the four participants in the pattern, namely, *Subject, ConcreteSubject, Observer, and ConcreteObserver*. Before proceeding with the TLG specification of the observer pattern, the structure and the informal description of the participants and collaborations of the observer pattern, as given by Gamma et al. [Gamma et al. 1995], listed below. Also, the LePUS3 [Eden et al. 2007] specification of the observer pattern is included below since it gives a less ambiguous representation of the structure of the pattern compared to the UML representation of the structure of the observer pattern as given by Gamma et al. [Gamma et al. 1995].

Figure 4.1: Structure of the Observer Pattern [Gamma et al. 1995]

**Participants**:

- **Subject**:

  o Knows its observers. Any number of Observer objects may observe a

  subject.

  o Provides an interface for attaching and detaching Observer objects.

- **Observer**: defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteSubject**:

  o Stores state of interest to ConcreteObserver objects.

  o Sends a notification to its observers when its state changes.

- **ConcreteObserver**:

  o Maintains a reference to a ConcreteSubject object.

  o Stores state that should stay consistent with the subject's state.

  o Implements the Observer updating interface to keep its state consistent with the subject's state.

**Collaborations**:

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.

- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

There are four participants in the observer pattern: Subject, concreteSubject, Observer, and concreteObserver. Among the four participants, concreteSubject and concreteObserver are concrete classes, subject is an abstract class, and Observer is an interface. Each participant has its own methods. These four participants are described in more detail, along with their TLG representations, in the four subsections that follow.
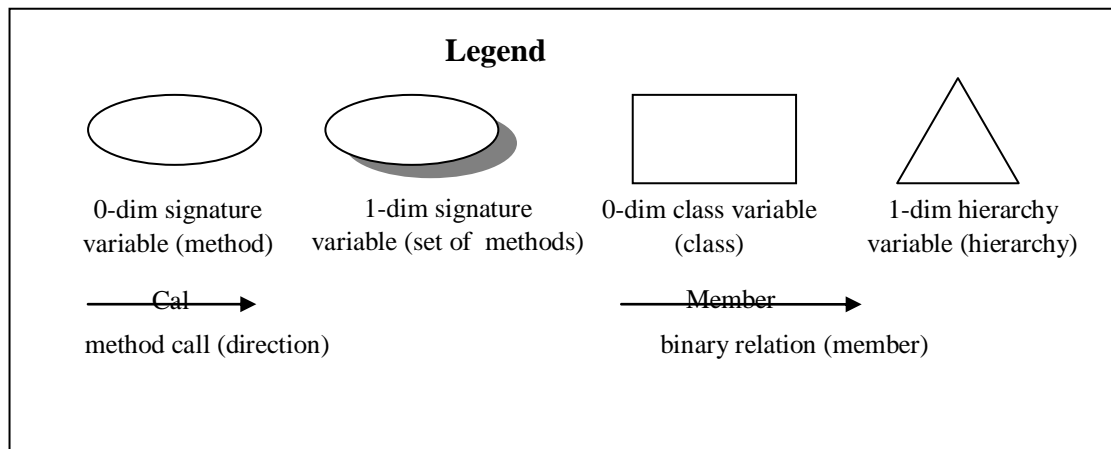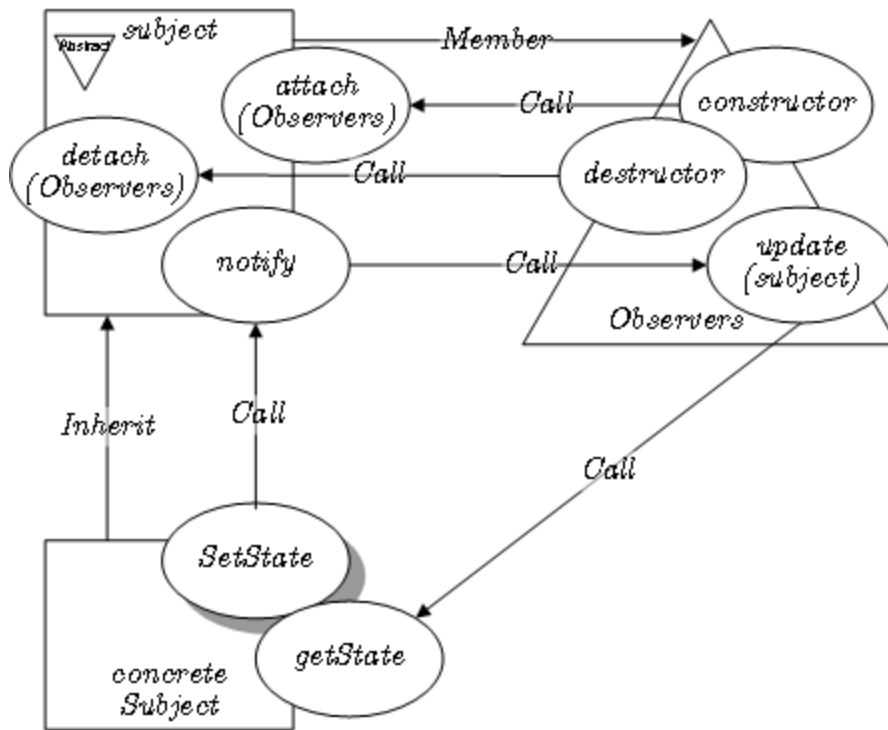
Figure 4.2: LePUS3 Specification of the Observer Pattern [Eden et al. 2007]

4.2.1 Abstract Class Subject

Based on the specification of the observer pattern given in Section 2.2 in Chapter II, *Subject* is an abstract class and it can be inherited by any number of subclasses to extend its functionalities provided. *Subject* has three methods that are not abstract (see Figure 4.1 and Figure 4.2). The abstract class should have a variable to keep the list of

observes that are observing any concrete subject (a concrete subject is a class that extends the abstract *Subject)*. Whenever a new observer starts to observe a concrete subject, it is added to a list inside the concrete subject. This list can be specified in TLG as follows:

List observer_list :: (concreteObservers)$^*$.

Each subject can be observed by zero or more observers. Whenever an observer is created based on a subject or whenever an observer is set to observe a subject, the observer gets attached to the subject and the subject keeps track of the observers using the collection of observer objects (i.e., a list of observer objects) that it has.

The Subject abstract class defines three methods that are used to keep track of the observers that are observing the subject and that are also used to notify the changes to the observers. The attach method is used to attach the observer that is created based on this subject. The detach method is used to detach the observer that is leaving the system or stops observing the subject. The notify method is used to notify all the observers, which are observing the current subject, about any changes that have happened to the subject.

Attach: observers_list.**Add**.
Detach: observers_list.**Remove**.
Notify: observers_list.**Update**.

Add, Remove, and Update are used to denote that observer objects can be added, deleted, and updated to the observers_list. So, the TLG specification of the abstract class Subject could be given as follows:

**Class abstract** Subject

//meta-notion
List observers_list:: (concreteObserver)$^*$.

//hyper-rules
Attach: observers_list.**Add**.
Detach: observers_list.**Remove**.

Notify: observers_list.**Update**.
End **Class**


4.2.2 Observers Interface

The *Observers* interface defines methods that can be implemented by the concrete

observers that want to observe a subject. This interface defines three methods for the

concrete observers to implement: constructor, destructor, and update. Since interfaces do

not provide methods definitions, the Observer interface can be written in TLG as follows:

**Interface** Observer
//hyper-rules
Constructor: .
Destructor: .
Update: .
**End Interface**

A concrete observer that implements this Observer interface can specify the

functionalities to the methods. The three methods specified by the interface are: the

constructor method that attaches a concrete observer to the subject by calling the Attach

method of the subject, the destructor method that detaches the concrete observer from the

subject by calling the Detach method of the concrete subject; and update updates the

concrete observer object by calling the getState method of the concrete subject.


4.2.3 Concrete Observers

The specifications for the concrete observers are defined in the Observer

interface. So, whenever a concrete observer is created, it implement the specifications

given in the Observer interface. The structure of a class implementing the Observer

interface in TLG can be written as follows:

**Class** concreteObserver **implements** Observer
    //hyper-rules
        Constructor: //functionality of the constructor.
        Destructor: //functionality of the destructor.
        Update: //functionality of update function.

**End Class**

Each function of the concreteObserver should be called with respect to the appropriate subject that it is interested in. Concrete observers have a reference to the subject(s) they are observing and this is achieved by providing a meta-notion:

Subject subObject :: (concreteSubject)+.

Here, subObject represent the subject(s) that the concreteObserver is observing, and subObject is an object of the abstract class *Subject*. The objects of any concrete subject can be assigned to the object of type Subject since the concrete subjects inherit from Subject. A concrete observer can observe one or more subjects and, when there is a change to any or all of the subjects that the concrete observer is observing, the changes will be conveyed to the concrete observer. The concrete observers can also check for every certain interval with the concrete subjects for any changes in the state of the subject that the observer is interested in. Finally, any concreteObserver class in TLG will look like:

**Class** concreteObserver **implements** Observer
    //meta-notions
        Subject subObject:: (concreteSubject)$^+$
    //hyper-rules
        Constructor: subObject.Attach.
        Destructor: subObject.Detach.
        Update: subObject.getState.
**End Class**

4.2.4 Concrete Subject

Based on the specification given in Section 2.2 and Chapter III, the interface for the concrete subjects is given by the abstract class Subject. So, the concrete classes can use the functions defined in the abstract class Subject instead of defining their own methods. In addition to the methods defined in Subject, the concrete subjects can also have two more methods of their own: setState and getState. These two methods are the accessors and mutators of the concrete subject class. In addition to the functionality of a mutator, the setState methods also perform the following: when there is a change made to the current state of the subject, setState is used to update the state of the subject and also to notify the observers, which are observing the subject, about the changes by using the Notify method defined in the abstract class.

The TLG specification of the concreteSubject is given below:

**Class** concreteSubject **extends** Subject
        //hyper-rules
            setState: this.Notify.
            getState: this.currentState.
**End Class**

The following section lists out the complete TLG specification of the observer pattern i.e., TLG specification of all the four participants (Subject, concreteSubject, Observer, and concreteObserverr) of the observer pattern.

**Class abstract** Subject

        //meta-notions
            List observers_list:: (concreteObserver)$^{*}$.

        //hyper-rules
            Attach: observers_list.**Add**.
            Detach: observers_list.**Remove**.
            Notify: observers_list.**Update**.

91

**End Class**

**Class** concreteSubject **extends** Subject
  //hyper-rules
    setState: this.Notify.
    getState: this.currentState.
**End Class**

**Interface** Observer
  //hyper-rules
    Constructor: .
    Destructor: .
    Update: .
**End Interface**

**Class** concreteObserver **implements** observer
  //meta-notions
    Subject subObject:: (concreteSubject)$^{+}$.
  //hyper-rules
    Constructor: subObject.Attach.
    Destructor: subObject.Detach.
    Update: subObject.getState.
**End Class**

One of the objectives of using the TLG is formal specification scheme was to be able to represent different levels of abstraction, i.e., to be able to capture and represent the different levels of abstraction involved in the concrete realization of the patterns. The TLG specification given above represents the core of the observer pattern. When additional details have to be implemented, more hyper- and meta-rules can be added to the specification given above to make it more concrete. For example, in the specification of the participant concreteObserver, additional predicates can be included to the hyper-rule update such as the following predicate:

Update: where currentSubjectState!=oldSubjectState, concreteObserver.setState.

Here, currentSubjectState represents the current state of the subject(s) that the observer is observing and oldSubjectState represents the state of the subject that was observed previously.

So, when the update hyper-rule is invoked, the predicate where currentSubjectState!=oldSubjectState is checked and, upon satisfaction of the predicate, control can be passed to the next function, i.e., concreteObserver.setState. If the predicate where currentSubjectState!=oldSubjectState fails, the control will not be passed to concreteObserver.setState. Here, setState is the mutator of concreteObserver.

The concreteObsever class with the predicate *where* in the *Update* hyper-rule is given below:

**Class** concreteObserver **implements** observer
  //meta-notions
    Subject subObject:: (concreteSubject)$^+$
    oldSubjectState:: subObject.getState.
    currentSubjectState:: subObject.getState;
  //hyper-rules
    Constructor: subObject.Attach.
    Destructor: subObject.Detach.
    Update: where currentSubjectState!=oldSubjectState,
      concreteObserver.setState; .
**End Class**

Other details can be added to the specification by including additional hyper- and meta-rules to the existing core specification to represent different levels of abstraction involved in the concrete realization of the observer pattern or, in general, other patterns.

# CHAPTER V

# SUMMARY, CONCLUSIONS, AND FUTURE WORK

5.1 Summary

Chapter I discusses the disadvantages of natural language representation of design patterns, followed by how those disadvantages can be overcome by using formal specification schemes. This chapter contains a brief discussion of the drawbacks of some of the existing formal specification schemes.

Chapter II provides background knowledge on design patterns as well as detailed descriptions of some of the existing formal schemes used to formalize patterns. This chapter also elaborates on the disadvantages of natural-language representation of design patterns by providing the GoF representation of the observer pattern. A description of some existing formal specification schemes along with their representation of the observer pattern is also included in Chapter III followed by a comparison of three existing specification schemes. Overall, this chapter provides justification for another formal specification scheme.

Chapter III describes the use of TLG as a formal specification language for object-oriented design patterns. This chapter provides information on how the building

blocks of object-oriented software systems can be represented in TLG.

Chapter IV provides a representation of the observer pattern in TLG. It elaborates on how each participant of the observer pattern can be represented in TLG.

5.2 Conclusions

TLG was used as a formal specification language to capture and represent the structural aspects of design patterns. It has been demonstrated that the TLGs have the capability to represent the building blocks of object-oriented software systems. The primary advantage of TLGs in defining design patterns is that specifications written in TLGs are understandable due to the natural-language-like vocabulary [Bryant et al. 1986] [Edupuganty 1987] [Lee 2003] [Maluszynski 1984]. TLGs could help pattern users understand the formalized version of patterns more readily compared to other formal specification methods that are difficult to understand due to their arcane mathematical notations.

5.3 Future Work

This work offers a number of possible future research directions to the software pattern community ranging from formal analysis of the behavioral elements of patterns through probably automatic implementation of patterns.

The behavioral semantics of design patterns could be analyzed to capture the key properties exhibited by each design pattern in order to validate the two-level representation of each pattern and in order to test whether or not the two-level grammar representation captures all the key aspects of design patterns. A classification scheme

based on the key properties of patterns could be developed and a process to implement this scheme to form a pattern catalog system could be provided. An algorithm could be developed to automatically implement design patterns from their TLG representations. The entire process, from formal specification of the structural elements of patterns to their implementation, could be a tool, which would generate an intermediate representation of design patterns form their two-level grammar representation, on the way to automatic realization of patterns.

# REFERENCES

[Agerbo and Cornils 1998] Ellen Agerbo and Aino Cornils, "How to Preserve the Benefits of Design Patterns", *Proceedings of the 13th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1998),* pp. 134-143, Vancouver, British Columbia, Canada, October 1998.

[Alexander et al. 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Schlomo Angel, *A Pattern Language*, Oxford University Press, New York, NY, 1977.

[Alexander 1979] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, New York, NY, 1979.

[Arnold and Gosling 1996] Ken Arnold and James Gosling, *The Java Programming Language,* Addison-Wesley Publishing Company, Reading, MA, 1996.

[Bayley and Zhu 2008] Ian Bayley and Hong Zhu, "Specifying Behavioral Features of Design Patterns in First Order Logic", *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pp. 203-210, Turku, Finland, August 2008.

[Beverly et al. 2004] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill, *Patterns for Parallel Programming*, Addison-Wesley Professional, Boston, MA, 2004.

[Buschmann et al. 2007] Frank Buschmann, Kevlin Henney, and Douglas Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, John Wiley & Sons, West Sussex, England, May 2007.

[Bosch 1996] Jan Bosch, "Language Support for Design Patterns", *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS) Conference,* pp. 197-210, Paris, France, February 1996.

[Bosch 1998] Jan Bosch, "Design Patterns as Language Constructs", *Journal of Object Oriented Programming,* Vol. 11, No. 2, pp. 18-52, May 1998.

[Bryant and Pan 1992] Barrett R. Bryant and Aiqin Pan, "Two-Level Grammar: A Functional/Logic Query Language for Database and Knowledge-Base Systems", *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR),* pp. 78-83, St. Petersburg, Russia, July 1992.

[Bryant et al. 1986] Barrett R. Bryant, Balanjaninath Edupuganty, William S. Chao, and Danny C. Deng, "Two-Level Grammar as a Programming Language for Data Flow and Pipelined Algorithms", *Proceedings of the International Conference on Computer Langugages,* pp. 136-143, Miami Beach, Florida, October1986.

[Buschmann et al. 2007] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt, "Past, Present, and Future Trends in Software Patterns", *IEEE Software*, Vol. 24, No. 7, pp. 31-37, August 2007.

[Buschmann et al. 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Inc., New York, NY, 1996.

[Caol et al. 2002] Fei Cao1, Barrett R. Bryant, Rajeev R. Raje, Mikhail Auguston, Andrew M. Olson, and Carol C. Burt, "Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar Using Domain Specific Knowledge", *Lecture Notes in Computer Science: Formal Methods and Software Engineering,* pp. 103-107, Springer-Verlag, London, UK, 2002.

[Chinnasamy 2000] Sivakumar Chinnasamy, "ELePUS: Extended Language for Pattern Uniform Specification", Master of Science Thesis, Department of Computer Science, Perdue University, West Lafayette, IN, August 2000.

[Cleaveland and Uzgalis 1977] J. C. Cleaveland and R. C. Uzgalis, *Grammars for Programming Languages*, Elsevier North-Holland, New York, NY, 1977.

[Cline 1996] Marshall P. Cline, "The Pros and Cons of Adopting and Applying Design Patterns in the Real World", *Communications of the ACM*, Vol. 39, No. 10, pp. 47-49, October 1996.

[Coplien 1991] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, 1991.

[De Graaf and Ollongran 1984] J. De Graaf and A. Ollongran, "On Two-Level Grammars", *International Journal of Computer Mathematics*, Vol. 15, No. 3-4, pp. 269-288, 1984.

[Dong 2002] Jing Dong, "UML Extensions of Design Pattern Compositions", *Journal of Object Technology,* Vol. 1, No. 5, pp. 149-161, November 2002.

[Eden 2002] Amnon H. Eden, "A Theory of Object-Oriented Design", *Information Systems Frontiers*, Vol. 4, No. 4, pp. 379-391, December 2002.

[Eden 2000] Amnon H. Eden, "Precise Specification of Design Patterns and Tool Support in Their Application", Doctoral Dissertation, Department of Computer Science, Tel Aviv University, Tel Aviv, Israel, 2000.

[Eden 1998] Amnon H. Eden, "Giving 'The Quality' a Name", *Journal of Object-Oriented Programming*, Vol. 11, No. 3, pp. 5-11, June 1998.

[Eden and Gasparis 2009] Amnon H. Eden and Epameinondas Gasparis, "Three Controlled Experiments in Software Engineering with the Two-Tier Programming Toolkit: Final Report*", Technical report CES-496*, School of Computer Science and Electronic Engineering, University of Essex, Colchester, United Kingdom, 2009. URL: http://ttp.essex.ac.uk/main/experiment/, date created: May 2009, date accessed: September 2009.

[Eden and Hirshfeld 2001] Amnon H. Eden and Yoram Hirshfeld, "Principles in Formal Specification of Object-Oriented Design and Architecture", *Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, p. 3, Toronto, Canada, November 2001.

[Eden et al. 2007] Amnon H. Eden, Epameinondas Gasparis, and Jonathan Nicholson, "LePUS3 and Class-Z Reference Manual", *Technical  Report CSM-474*, School of Computer Science and Electronic Engineering , University of Essex, Colchester, United Kingdom, 2007. URL: http://lepus.org.uk/ref/refman/refman.xml, date created: December 2007, date accessed: September 2009.

[Edupuganty 1987] Balanjaninath Edupuganty, "Two-Level Grammar: An Implementable Metalanguage for Consistent and Complementary Language Specifications", Doctoral Dissertation, Department of Computer Science, University of Alabama, Birmingham, Alabama, 1987.

[Endrei 2004] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pal Krogdahl, Min Luo, and Tony Newling, *Patterns: Service-Oriented Architecture and Web Services*, International Business Machines Corporation (IBM) Publication, Armonk, New York, NY, 2004.

[Fowler 1997] Martin Fowler, *Analysis Patterns,* Addison-Wesley Publishing Company, Reading, MA, 1997.

[Fowler 2003] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Longman Publishing Co., Inc., Bostan, MA, 2003.

[France et al. 2004] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song, "A UML-Based Pattern Specification Technique", *IEEE Transactions on Software Engineering*, Vol. 30, No. 3, pp. 193-206, March 2004.

[Furtado et al. 1983] Antonio L. Furtado, Paulo A. S. Veloso, and Marco A. Casanova, "A Grammatical Approach to Data Bases", *Proceedings of the 9th International Federation for Information Processing (IFIP) World Computer Congress Conference,* pp. 701-710, Paris, France, September 1983.

[Gamma et al. 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[Goldberg and Robson 1989] Adele Goldberg and David Robson, *Smalltalk-80: The Language*, Addison-Wesley Professional, Boston, MA, 1989.

[Grune 1984] Dick Grune, "How to Produce All Sentences from a Two-Level Grammar", *Information Processing Letters*, Vol. 19, No. 4, pp. 181-185, March 1984.

[Helm et al. 1990] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Compositions in Object Oriented Systems", *Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 90),* pp. 19-180, Ottawa, Canada, October 1990.

[Hallstrom 2004] Jason O. Hallstrom, "Design Patterns Contracts", Doctoral Dissertation, Department of Computer Science, Ohio State University, Columbus, Ohio, 2004.

[Henninger and Corrêa 2007] Scott Henninger and Victor Corrêa, "Software Pattern Communities: Current Practices and Challenges", *Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP)*, pp. 1-19, Monticello, Illinois, September 2007.

[Hillside 1993] The Hillside Group, "The Hillside Group", URL: http://hillside.net/, date created: 1993, date accessed: August 2009.

[ISO] International Organization for Standardization, "International Standards for Business, Government, and Society", URL: http://www.iso.org/iso/home.htm, date created: unknown, date accessed: August 2009.

[Johnson 1997] Ralph E. Johnson, "Frameworks = (Components + Patterns)", *Communications of the ACM*, Vol. 40, No. 10, pp. 39-42, October 1997.

[Kent and Lauder 2004] Stuart Kent and Anthony Lauder, "Precise Visual Specification of Design Patterns", *Lecture Notes in Computer Science: 11th European Conference on Object-Oriented Programming (ECOOP 04)*, p. 114, Berlin, Germany, 2004.

[Kim and Carrington 2004] Soon-Kyeong Kim and David Carrington, "Using Integrated Meta-Modeling to Define Object-Oriented Design Patterns with Object-Z and UML", *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004),* pp. 257-264, Busan, Korea, December 2004.

[Klarlund et al. 1996] Nils Klarlund, Jari Koistinen, and Michael I. Schhwatbach, "Formal Design Constraints", *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 96),* pp. 114-134, San Jose, California, October 1996.

[Konrd et al. 2004] Sascha Konrd, Betty H. C. Cheng, and Laura A. Campbell, "Object Analysis Pattern for Embedded Systems", *IEEE Transactions on Software Engineering*, Vol. 30, No. 12, pp. 970-992, December 2004.

[Kvale 1996] Steinar Kvale, *Interviews: An Introduction to Qualitative Research Interviewing*, Sage Publications, Inc., Thousand Oaks, CA, 1996.

[Lauder and Kent 1998] Anthony Lauder and Stuart Kent, "Precise Visual Specification of Design Patterns", *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP),* pp. 114-134, Brussels, Belgium, July 1998.

[Lee 2003] Beum-Seuk Lee, "Automated Conversion from a Requirements Document to an Executable Formal Specification Using Two-Level Grammar and Contextual Natural Language Processing", Doctoral Dissertation, Department of Computer Science, University of Alabama, Birmingham, Alabama, 2003.

[Lieberman 1986] Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1986),* pp. 214-223, Portland, Oregon, November 1986.

[Liu et al. 2005] Shih-Hsi Liu, Fei Cao, Barrett R. Bryant, Jeff Gray, Rajeev R. Raje, Andrew M. Olson, and Mikhail Auguston, "Quality of Service-Driven Requirements Analyses for Component Composition: A Two-Level Grammar++ Approach", *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pp. 731-734, Howard International House, Taipei, Taiwan, July 2005.

[Maluszynski 1984] Jan Maluszynski, "Towards a Programming Language Based on the Notion of Two-Level Grammar", *Theoretical Computer Science*, Vol. 28, No. 9, pp. 13-43, December 1984.

[Mak et al. 2004] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun, "Precise Modeling of Design Patterns in UML", *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004),* pp. 252-261, Washington, DC, May 2004.

[Mikkonen 1995] Tommi Mikkonen, "Partitioning DisCo Specifications", *Proceedings of the IEEE Colloquium on Partitioning in Hardware-Software Codesigns*, pp. 6/1-6/8, London, United Kingdom, February 1995.

[Mikkonen 1998] Tommi Mikkonen, "Formalizing Design Patterns", *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pp. 115-124, Kyoto, Japan, April 1998.

[Palay et al. 1988] A. J. Palay, W. J. Hansen, M. L. Kazar, M. Sherman, M. G. Wadlow, T. P. Neuendorffer, Z. Stern, M. Bader, and T. Peters, "The Andrew Toolkit- An Overview", *Proceedings of the USENIX Winter Conference*, pp. 9-21, Dallas, Texas, January 1988.

[Rising 2000] Linda Rising, *The Pattern Almanac 2000,* Addison-Wesley Publishing Company, Reading, MA, 2000.

[Saacks and Hassell 1989] Marguerite Saacks and Johnette Hassell, "Two-Level Grammars as a Technique for Formalizing Programming Schemes", *Proceedings of the 17th ACM Annual Computer Science Conference*, pp. 305-308, Louisville, Kentucky, February 1989.

[Sabatucci et al. 2009] Luca Sabatucci, Massimo Cossentino, and Angelo Susi, "Introducing Motivations in Design Pattern Representation", *Lecture Notes in Computer Science: Proceedings of the 11th International Conference on Software Reuse*, pp. 201-210, Springer-Verlag, London, UK, 2009.

[Schmidt 1988] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, William C Brown Publishers, Dubuque, IA, 1988.

[Schmid and Verlage 2002] Klaus Schmid and Martin Verlage, "The Economic Impact of Product Line Adoption and Evolution", *IEEE Software*, Vol. 19, No. 4, pp. 50-57, July 2002.

[Sintzoff 1967] Michel Sintzoff, "Existence of a van Wijngaarden Syntax for Every Recursively Enumerable Set", *Annals of the Scientific Society of Brussels,* Vol. 8, No. 2, pp. 115-118*,* June 1967.

[Soukup 1995] Jiri Soukup, *Implementing Patterns in Pattern Languages of Program Design*, Addison-Wesley Publishing Co., Inc., New York, NY, 1995.

[Soundarajan and Hallstrom 2004] Neelam Soundarajan and Jason O. Hallstrom, "Responsibilities and Rewards: Specifying Design Patterns", *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pp. 666-675, Edinburgh, Scotland, May 2004.

[Spivey 1998] J. M. Spivey, *The Z Notation: a Reference Manual*, Prentice Hall International (UK) Ltd., Programming Research Group, University of Oxford, Oxford, UK, 1998. URL: http://www.rose-hulman.edu/class/csse/cs415/zrm.pdf, date created: 1998, date accessed: June 2010.

[Stroustrup 1991] Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley Publishing Company, Reading, MA, 1991.

[Taibi and Taibi 2006] Toufik Taibi and Fathi Taibi, "Formal Specification of Design Patterns and Their Instances", *Proceedings of the International Conference on Computer Systems and Applications (ICCSA 2006)*, pp. 33-36, Sharjah, UAE, March 2006.

[Taibi 2006] Toufik Taibi, "Formalizing Design Patterns Composition", *IEEE Software*, Vol. 153, No. 3, pp. 127-136, June 2006.

[Taibi and Ling 2003 A] Toufik Taibi and David Ngo Chek Ling, "Formal Specification of Design Patterns: A Comparison", *Proceedings of the International Conference on Computer Systems and Applications (ICCSA 2003)*, p. 77, Tunisia, Morocco, July 2003.

[Taibi and Ling 2003 B] Toufik Taibi and David Ngo Chek Ling, "Formal Specification of Design Patterns – A Balanced Approach", *Journal of Object Technology*, Vol. 2, No. 4, pp. 127-140, August 2003.

[Taibi 2007] Toufik Taibi, *Design Pattern Formalization Techniques*, Idea Group Inc., Hershey, PA, 2007.

[Cleaveland and Uzgalis 1973] J. C. Cleaveland and R. C. Uzgalis, "What Every Programmer Should Know About Grammar", *Modelling and Measurement Notes,* No. 12, Department of Computer Science, University of California, Los Angeles, California, March 1973.

[van Wijngaarden 1965] A. van Wijngaarden, "Orthogonal Design and Description of a Formal Language", *Technical Report*, Mathematisch Centrum, Amsterdam, October 1965.

[van Wijngaarden et al. 1975] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. T. Meertens, and R. G. Fisker, "Revised Report on the Algorithmic Language ALGOL68", *Acta Informatica*, pp. 1-236, May 1975.

[Vlissides and Linton 1990] John M. Vlissides and Mark A. Linton, "Unidraw: A Framework for Building Domain-Specific Graphical Editors", *ACM Transactions on Information Systems*, Vol. 8, No. 3, pp. 237-268, July 1990.

[Zhao 2006] Wei Zhao, "Model-Driven Integration of Software and Service Components", Doctoral Dissertation, Department of Computer Science, University of Alabama at Birmingham, Birmingham, Alabama, 2006.

APPENDICES

# APPENDIX A:GLOSSARY

Abstract Class      A class whose primary purpose is to define an interface. It defers some or all of its implementation to subclasses. An abstract class cannot be instantiated [Gamma et al. 1995].

Action      A syntactic unit of execution in DisCo which consists of a name, a list of participants and parameters, a guard, and a body. Actions are disjoint and atomic. An action can be executed when it is enabled. The execution of an action can only change the states of the participating objects. In a logical sense, an action is a relation between two adjacent states in an infinite sequence of states [Mikkonen 1998].

BPSL      Balanced Pattern Specification Language [Taibi 2006] is a formal specification language that attempts to formalize both the structural and behavioral aspects of design patterns using a subset of First Order Logic (FOL) and a subset of Temporal Logic of Actions (TLA).

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 [Gamma et al. 1995].

Concrete Class      A class having no abstract operations. Contrary to an abstract class, a concrete class can be instantiated.

Constructor      In object-oriented programming languages, a constructor is an operation that is automatically invoked to initialize new object instances.

Design Pattern      A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context [Gamma et al. 1995].

| | |
|---|---|
| Destructor | In object-oriented programming languages, a destructor is an operation that is automatically invoked to finalize an object that is about to be deleted [Gamma et al. 1995]. |
| DisCO | Distributed Cooperation [Mikkonen 1998] is an object-oriented specification language for specifying the behavioral aspects of reactive systems. |
| eLePUS | Extended LanguagE for Pattern Uniform Specification [Eden et al. 2007] is an extension of LePUS (see entry for LePUS3). |
| Fairness | Liveness properties are obtained by stating fairness requirements. In DisCo, fairness requirement indicates that it is not possible for an action to be enabled infinitely often without being executed infinitely often [Mikkonen 1998]. |
| Guard | Every DisCo action has a boolean expression called the guard. If there exist participants and parameters so that the guard evaluates to true, the action is said to be enabled [Mikkonen 1998]. |
| Implementation Overhead | When used in the context of programming languages, pattern users are required to implement a pattern again and again in different systems because classes and objects of patterns represented using programming languages are tightly coupled with other functionalities of the system. |
| Inheritance | A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or (in object-oriented programming languages) a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations [Gamma et al. 1995]. |
| Instance Variable | A piece of data that defines part of an object's representation [Gamma et al. 1995]. |
| Interaction Diagram | A diagram that shows the flow of requests among objects [Gamma et al. 1995]. |

| | |
|---|---|
| Interface | The set of all signatures defined by an object's operations. The interface describes the set of requests to which an object can respond [Gamma et al. 1995]. |
| LePUS3 and Class-Z | LanguagE for Pattern Uniform Specification 3 and Class-Z are object-oriented Design Description Languages (DDL) that are intended to abstract, model, and formalize object-oriented programs, design patterns, and application frameworks. LePUS3 is an extension of LePUS. |
| Liveness Property | A property of a potentially infinite execution which is of the form "something good will eventually occur" [Mikkonen 1998]. |
| Object | A run-time entity that packages both data and the procedures that operate on that data [Gamma et al. 1995]. |
| Overlap | An overlap between two patterns P1 and P2 means that there exists at least one element in pattern P1 which is also in pattern P2. |
| Overriding | Redefining an operation (inherited from a parent class) in a subclass [Gamma et al. 1995]. |
| Parent Class | The class from which another class inherits in object-oriented programming languages. |
| Participant | Execution of an action needs object(s) to participate in it. The number of participants and their classes are indicated in the action definition [Gamma et al. 1995]. |
| Pattern | A pattern is a named description of a problem, a solution, when to apply the solution, and how to apply the solution in new contexts. |
| Reactive System | Reactive system is a system that is in constant interaction with its environment [Mikkonen 1998]. |
| Reusability | When used in the context of programming languages, design patterns are represented at the implementation level, i.e., they are represented as classes and objects. Since these classes and objects are typically associated with other functionalities of the systems, it is hard to reuse the design pattern again without modification. |
| Self Problem | The implementation of several design patterns requires forwarding messages from an object receiving a message to an object implementing the behavior that is to be executed in response to the message. The receiving object can, for |

example, be an application domain object that delegates some messages to a strategy object. However, once the message is forwarded, the reference to the object originally receiving the message may no longer be available and the references to self refer to the delegated object, rather than to the original receiver of the message. This problem is known as the self problem [Lieberman 1986].

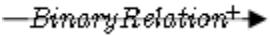| | |
|---|---|
| Semantics | The assignment of meaning to various entites. |
| Traceability | The traceability of a design pattern is often lost between classes and objects, i.e., the pattern, which is a conceptual entity at the design level, is scattered over different parts of an object or even multiple objects. This problem was first identified by Soukup [Soukup 1995]. |
| UML | Unified Modeling Language, a standard notation for modeling systems using object-oriented concepts. |
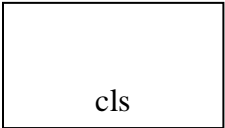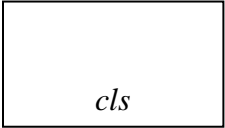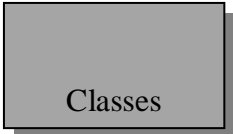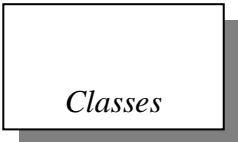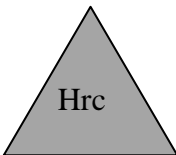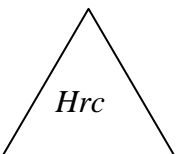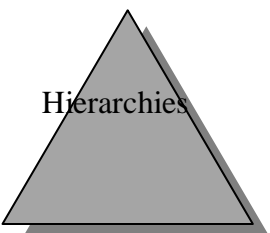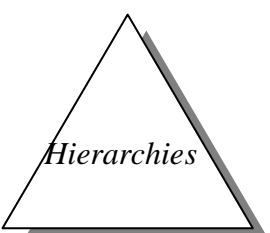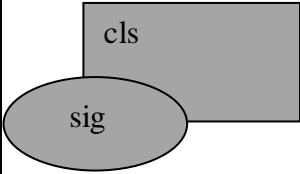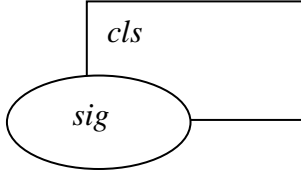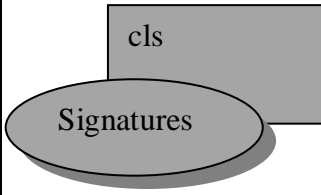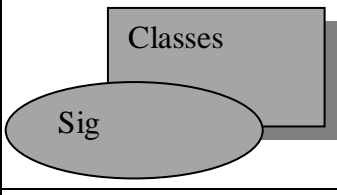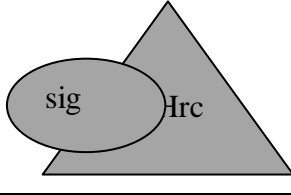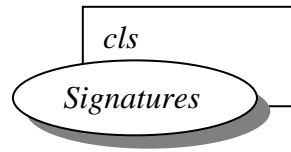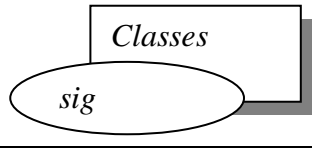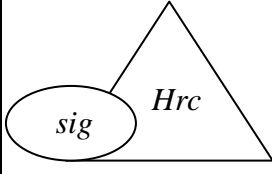
# APPENDIX B

## Terms in LePUS3 and Class-Z
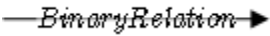
This appendix contains the symbols and terms used in the LePUS3 and Class-Z

specification scheme [Eden et al. 2007].

| Type | Symbol in LePUS3 | Symbol in Class-Z | Symbol Name |
|---|---|---|---|
| **CLASS** (a class) | cls | cls | 0-dimensional class constant |
| | *cls* | *cls* | 0-dimensional class variable |
| **℘CLASS** (set of classes) | Classes | Classes | 1-dimensional class constant |
| **℘CLASS** (a set of classes) | *Classes* | *Classes* | 1-dimensional class variable |

| | | | |
|---|---|---|---|
| SIGNATURE (a method declaration) | sig | sig | 0-dimensional signature constant |
| | *sig* | *sig* | 0-dimensional signature variable |
| PSIGNATURE (a set of method declarations) | Signatures | Signatures | 1-dimensional signature constant |
| | *Signatures* | *Signatures* | 1-dimensional signature variable |
| HIERARCHY (a set of classes which contains one class such that all other classes inherit (possibly indirectly) from it) | Hrc | Hrc | 1-dimensional hierarchy constant |
| | *Hrc* | *Hrc* | 1-dimensional hierarchy variable |
| PHIERARCHY (a set of hierarchies) | Hierarchies | Hrcs | 2-dimentional hierarchy constant |
| | *Hierarchies* | *Hrcs* | 2-dimentional hierarchy variable |

| | | | |
|---|---|---|---|
| **METHOD** (a method with signature sig which is a member of (or inherited by) class cls) | cls / sig (diagram) | $sig \otimes cls$ | 0-dimensional method constant term |
| | cls / sig (diagram) | $sig \otimes cls$ | 0-dimensional method variable terms |
| **PMETHOD** (A tribe (a set of methods with signatures Signatures ) that are members of (or inherited by) class cls) | cls / Signatures (diagram) | $Signatures \otimes cls$ | 1-dimensional superimposition (method) constant terms |
| | Classes / Sig (diagram) | $sig \otimes Classe$ | |
| | sig / Hrc (diagram) | $sig \otimes Hrc$ | |
| | cls / Signatures (diagram) | $Signatures \otimes cls$ | 1-dimensional superimposition (method) variable terms |
| | Classes / sig (diagram) | $sig \otimes Classes$ | |
| | sig / Hrc (diagram) | $sig \otimes Hrc$ | |

Relation Symbols in LePUS3 and Class-Z

| Symbol in LePUS3 | Symbol in Class-Z | Symbol name |
|---|---|---|
|  | UnaryRelation | Unary relation symbol |
|  | BinaryRelation | Binary relation symbol |
|  | BinaryRelation$^+$ | Transitive binary relation symbol |
|  | *ALL* | ALL predicate symbol |
|  | *TOTAL* | TOTAL predicate symbol |
|  | *ISOMORPHIC* | ISOMORPHIC predicate symbol |

Predicate Formulas in LePUS3 and Class-Z

| Predicate formulas in Class-Z | Predicate formulas in LePUS3 |
|---|---|
| ALL(UnaryRelation,T1) | An ALL predicate symbol marked with *UnaryRelation* placed over T1 |
| TOTAL(BinaryRelation,τ1,τ2) | A TOTAL predicate symbol marked with a *BinaryRelation* connecting τ1 to τ2 |
| ISOMORPHIC(BinaryRelation,T1,T2) | An ISOMORPHIC predicate symbol marked with a *BinaryRelation* connecting T1 to T2 |

# APPENDIX C

## TLG Specification of the GoF Design Patterns

This appendix contains the TLG specification of the GoF design patterns [Gamma et al. 1995] along with the descriptions of the pattern elements *intent*, *structure*, and *participant* of each GoF design patterns.

## Abstract Factory

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
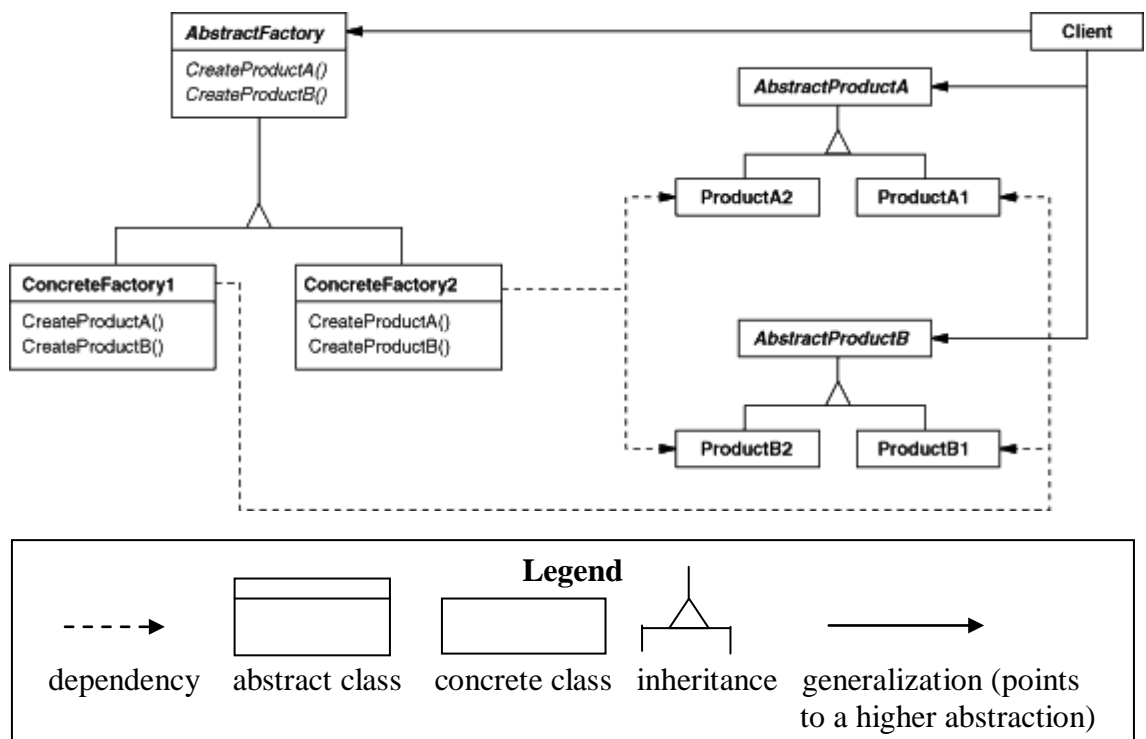
**Structure:**



Figure A-1: UML Representation of the Abstract Factory Pattern [Gamma et al. 1995]

**Participants:**

- AbstractFactory (WidgetFactory): 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): 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.

**TLG Specification of the Abstract Factory Pattern:**

```
Interface Factories
        FactoryProducts:.
End Interface

Products: Product-1; . . . ; Product-n.
Interface Products
        Product:.
End Interface

Class concreteFactory implements Factories
        Products: Product-1; . . . ; Product-n.
        Products: create and return concrete Products.
End Class

Class concreteFactoryProducts implements Products
        Product: specification of Product.
End Class
```

Factory Method

**Intent:** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Structure:**
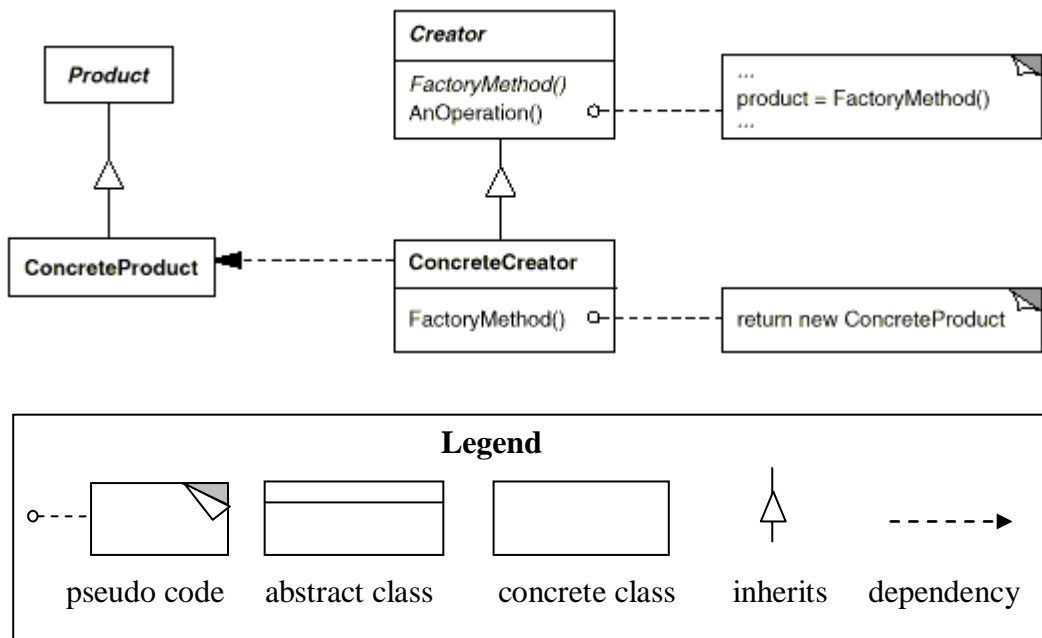


Figure A-2: UML Representation of the Factory Method Pattern [Gamma et al. 1995]

**Participants:**

- Product (Document): Defines the interface of the objects that the factory method creates.

- ConcreteProduct (MyDocument): Implements the Product interface.

- Creator (Application): Declares the factory method which returns an object of type Product: Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object, it may call the factory method to create a Product object.

- ConcreteCreator (MyApplication): Overrides the factory method to return an instance

  of a ConcreteProduct.

**TLG Specification of the Abstract Factory Method Pattern:**

Interface Creator
      FactoryMethod:.
End Interface

Class concreteCreator implements Creator
      concreteProduct :: concreteProduct-1; . . . ; concreteProduct-n.
      FactoryMethod : create and return concreteProduct.
End Class

Interface Product
      Ops:.
End Interface

concreteProduct :: concreteProduct-1; . . . ; concreteProduct-n.
Class concreteProduct implements Product
      Ops: create concreteProduct specific Ops.
End Class

Adapter

**Intent:** Convert the interface of a class into another interface as desired by the clients. Adapter lets classes work together that couldn't otherwise, because of incompatible interfaces.
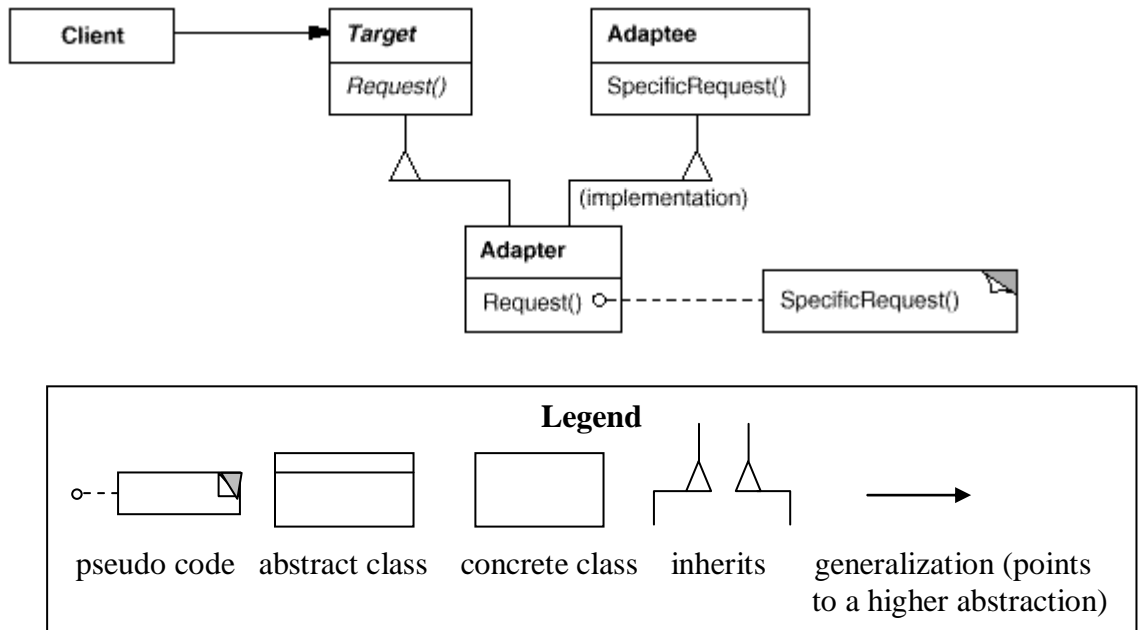
**Structure:**



Figure A-3: UML Representation of the Adapter Pattern [Gamma et al. 1995]

**Participants:**

- Target (Shape): Defines the domain-specific interface that Client uses.

- Client (DrawingEditor): Collaborates with objects conforming to the Target interface.

- Adaptee (TextView): Defines an existing interface that needs adapting.

- Adapter (TextShape): Adapts the interface of Adaptee to the Target interface.


**TLG Specification of the Adapter Pattern:**

Abstract Class target
        Requests: client specific requests.
End Class

Class Client
        Target target :: concreteTarget.
        Operations : target.Request.
End Class

Class adapter extends target, adaptee
        //The *Request* operation in *adapter* modifies the *Request* operation in the *target* to
        //make it reusable by the *adaptee*.
        Request: adaptee.SpecificRequests
End Class

Class adaptee
        SpecificRequests: adaptee specific requests.
End Class

Class concreteTarget extends Target
        //concreteTarget specific operations
End Class

Bridge

**Intent:** Decouple an abstraction from its implementation so that the two can vary independently.
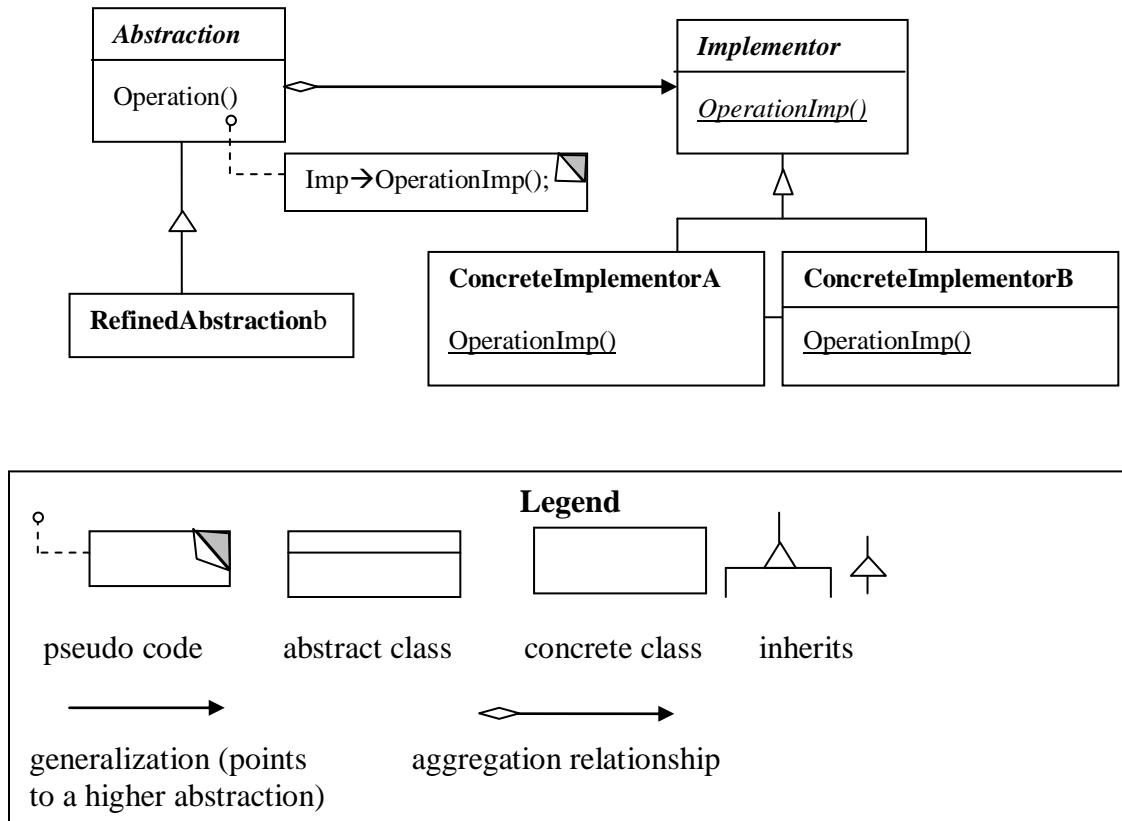
**Structure:**



Figure A-4: UML Representation of the Bridge Pattern [Gamma et al. 1995]

**Participants:**

- Abstraction (Window): Defines the abstraction's interface and maintains a reference to an object of type Implementor.

- RefinedAbstraction (IconWindow): Extends the interface defined by Abstraction.

- Implementor (WindowImp): Defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface. In fact, the two

interfaces can be quite different. Typically, the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

- ConcreteImplementor (XWindowImp, PMWindowImp): Implements the Implementor interface and defines its concrete implementation.

**TLG Specification of the Bridge Pattern:**

Abstract Class abstraction
        AbstractOps: concreteImplementations.ConcreteOps.
End Class

Class refinedAbstraction extends abstraction
        abstraction  concreteAbstraction :: concreteAbstarction-1; . . . ;
        concreteAbstarction-n.
        CompisiteOps: concreteAbstraction.AbstractOps.
End Class

Interface Implementations
        concreteOps:.
End Interface

Class concreteImplementations
        concreteOps:concreteImplementationsOps.
End Class

Composite

**Intent:** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
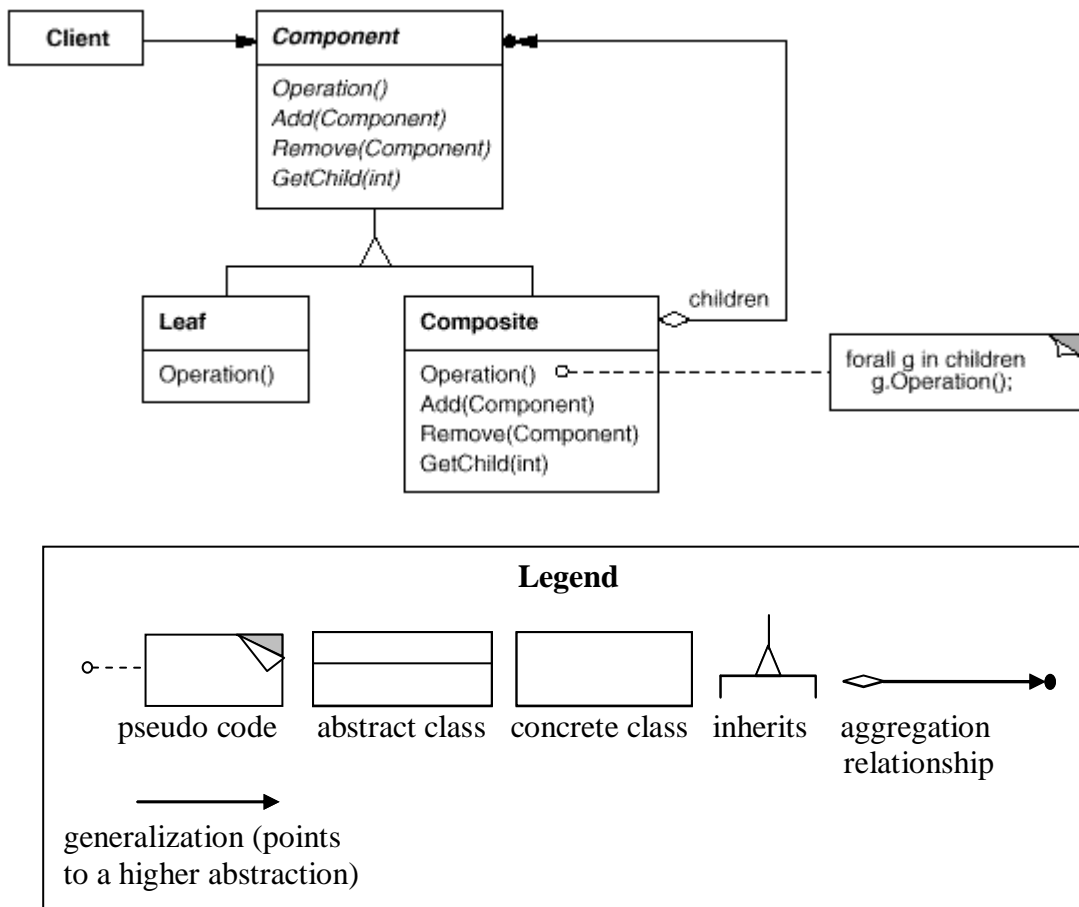
**Structure:**



Figure A-5: UML Representation of the Composite Pattern [Gamma et al. 1995]

**Participants:**

- Component (Graphic): Declares the interface for objects in the composition. Implements default behavior for the interface common to all classes, as appropriate. Declares an interface for accessing and managing its child components. Defines an

122

interface for accessing a component's parent in the recursive structure, and implements it if appropriate (optional).

- Leaf (Rectangle, Line, Text, etc.): Represents leaf objects in the composition. A leaf has no children and defines behavior for the primitive objects in the composition.

- Composite (Picture): Defines behavior for the components having children, stores child components, and Implements child-related operations in the Component interface.

- Client: Manipulates objects in the composition through the Component interface.

**TLG Specification of the Composite Pattern:**

```
Abstract  Class component
        componentOps: create and return component.
End Class

Class levaes extends component
        componentOps: create and return leaves component.
End Class

Class composite extends component
        Component:: (component)+.
        Leaves:: (leaves)+.
        compositeComponent :: Component; Leaves; Component, Leaves; Leaves,
                              Component.
        componentOps: create and return Component.
        compositeOps: create and return compositeComponent.
End Class
```

Decorator

**Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a

flexible alternative to subclassing for extending functionality.
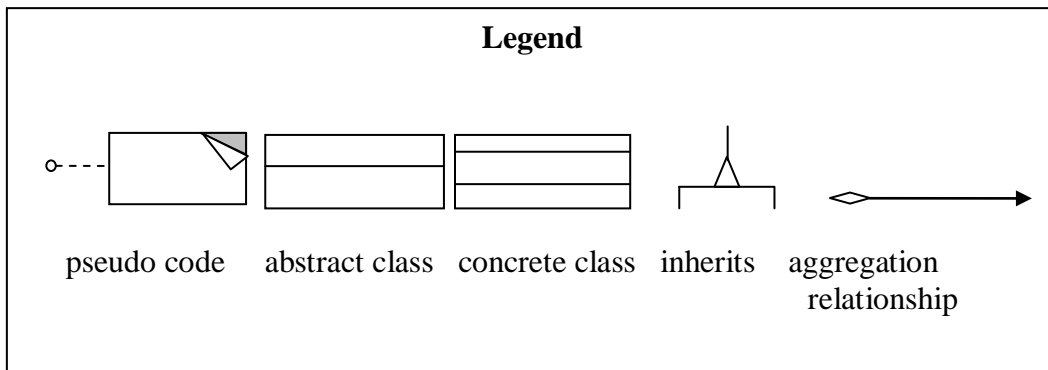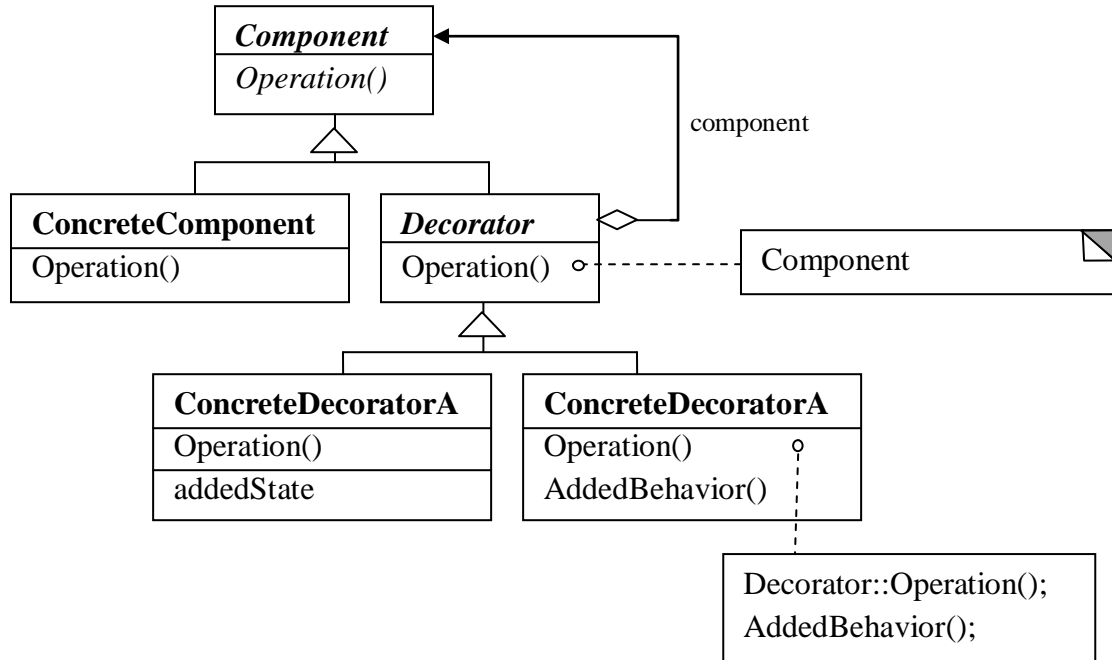
**Structure:**



Figure A-6: UML Representation of the Decorator Pattern [Gamma et al. 1995]

**Participants:**

- Component (VisualComponent): Defines the interface for objects that can have

  responsibilities added to them dynamically.

- ConcreteComponent (TextView): Defines an object to which additional responsibilities can be attached.

- Decorator: Maintains a reference to a Component object and defines an interface that conforms to the Component's interface.

- ConcreteDecorator (BorderDecorator, ScrollDecorator): Adds responsibilities to the component.

**TLG Specification of the Decorator Method Pattern:**

Abstract Class Component
        Abstract Ops:.
End Class

concreteComponents :: concreteComponent-1; . . . ; concreteComponenet-n.

Class concreteComponents extends Component
        Ops:ops specific to this concreteCompoenet.
End Class

Abstract Class Decorator extends Component
        Component component :: concreteComponents.
        Ops : component.Ops.
End Class

concreteDecorator :: concreteDecorator-1; . . . ; concreteDecorator-n.

Class concreteDecorator extends Decorator
        Ops: concreteDecorator.Ops.
End Class

Flyweight

**Intent:** Use sharing to support large numbers of fine-grained objects efficiently.
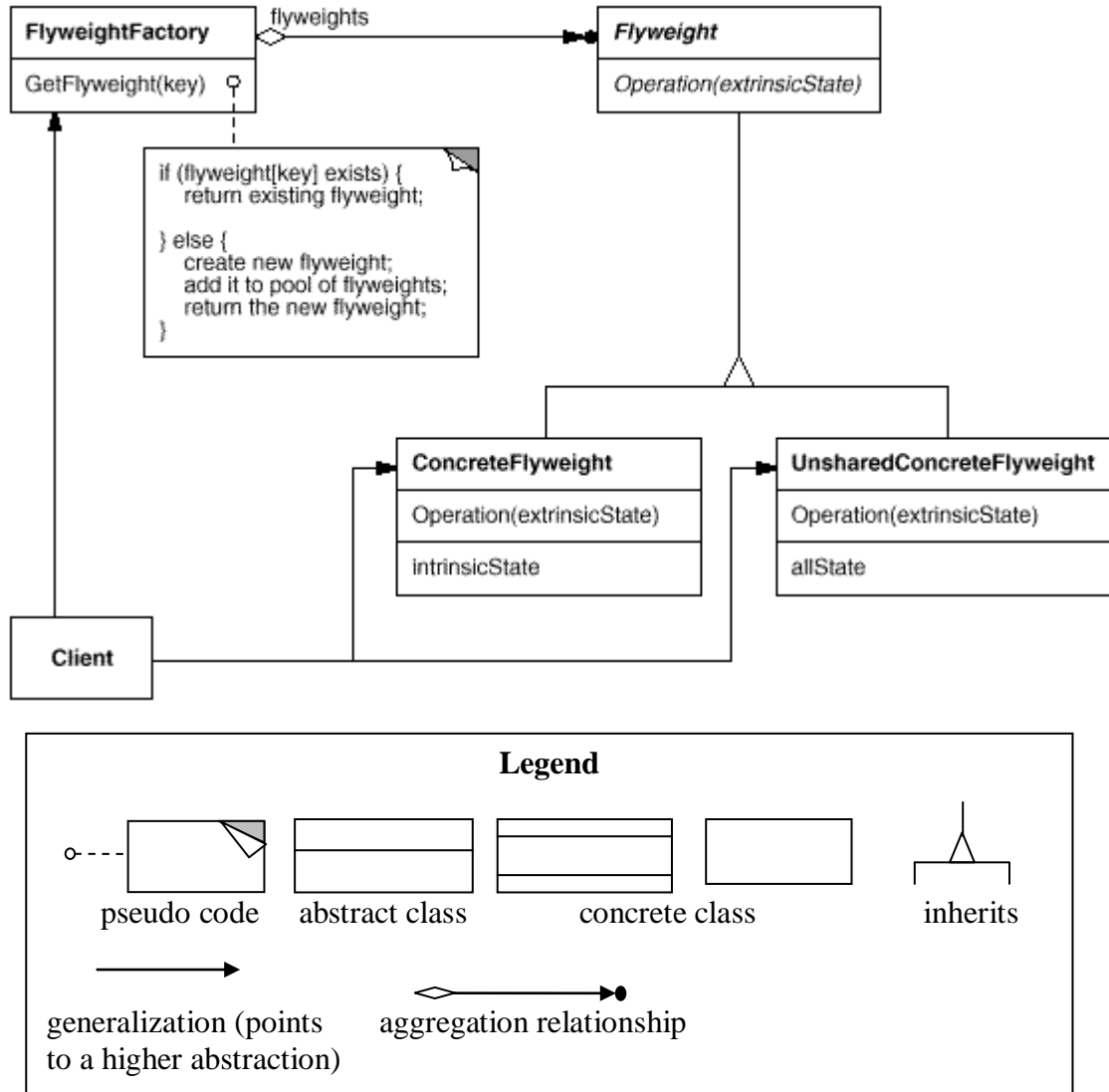
**Structure:**



Figure A-7: UML Representation of the Flyweight Pattern [Gamma et al. 1995]

**Participants:**

- Flyweight: Declares an interface through which flyweights can receive and act on extrinsic state.

- ConcreteFlyweight (Character): Implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.

- UnsharedConcreteFlyweight (Row, Column): Not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing, it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

- FlyweightFactory: Creates and manages flyweight objects. Ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one if none exists.

- Client: Maintains a reference to flyweight(s); Computes or stores the extrinsic state of flyweight(s).

**TLG Specification of the Flyweight Pattern:**

```
Interface FlyWeights
        extrinsicState::.
        extrinsicState Operation ::.
End Interface

Class concreteFlyWeight extends FlyWeights
        extrinsicState:: extrinsicState-1; . . . ; extrinsicState-n.
        extrinsicState Operation :: //some operation
End Class

Class flyweightFactory
        getFlyWeight : create and return a concreteFlyWeight.
End Class
Class Client
        Requests : flyWeightFactory.getFlyWeight.
End Class
```

Class unsharedConcreteFlyWeight
        intrinsicState:: extrinsicState-1; . . . ; extrinsicState-n.
        intrinsicState Operation :: //some operation
End Class

Proxy

**Intent:** Provide a surrogate or placeholder for another object to control access to it.
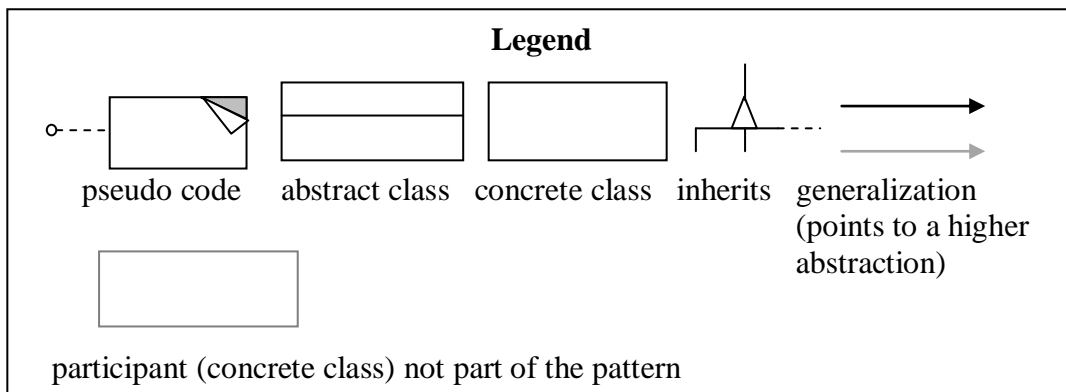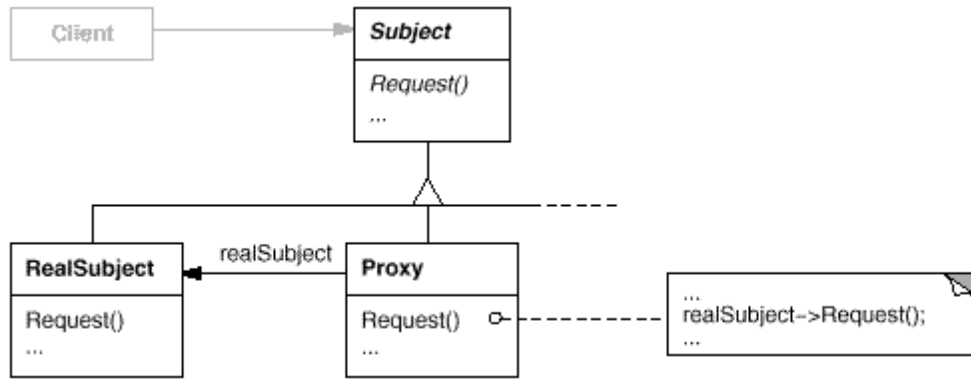
**Structure:**



Figure A-8: UML Representation of the Proxy Pattern [Gamma et al. 1995]

**Participants:**

- Proxy (ImageProxy): Maintains a reference that lets the proxy access the real subject, Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same. Proxy also provides an interface identical to the Subject's interface so that a proxy can be substituted for the real subject. A proxy controls access to the real subject and may be responsible for creating and deleting it.

- Subject (Graphic): Defines the common interface for RealSubject and Proxy so that a

- Proxy can be used anywhere a RealSubject is expected.

- RealSubject (Image): Defines the real object that the proxy represents.

**TLG Specification of the Proxy Pattern:**

Abstract Class Subject
        Request:
End Class

Class Proxy extends Subject
        Subject realSubject :: realSubject-1; . . . ; realSubject-n.
        Request : realsubject.Request.
End Class

realSubject :: realSubject-1; . . . ; realSubject-n.

Class realSubject extends Subject
        Request :: //some ops.
End Class

Class Client
        Subject concreteSubject :: realSubject-1; . . . ; realSubject-n.
        Ops : concreteSubject.Request.
End Class

Iterator

**Intent:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
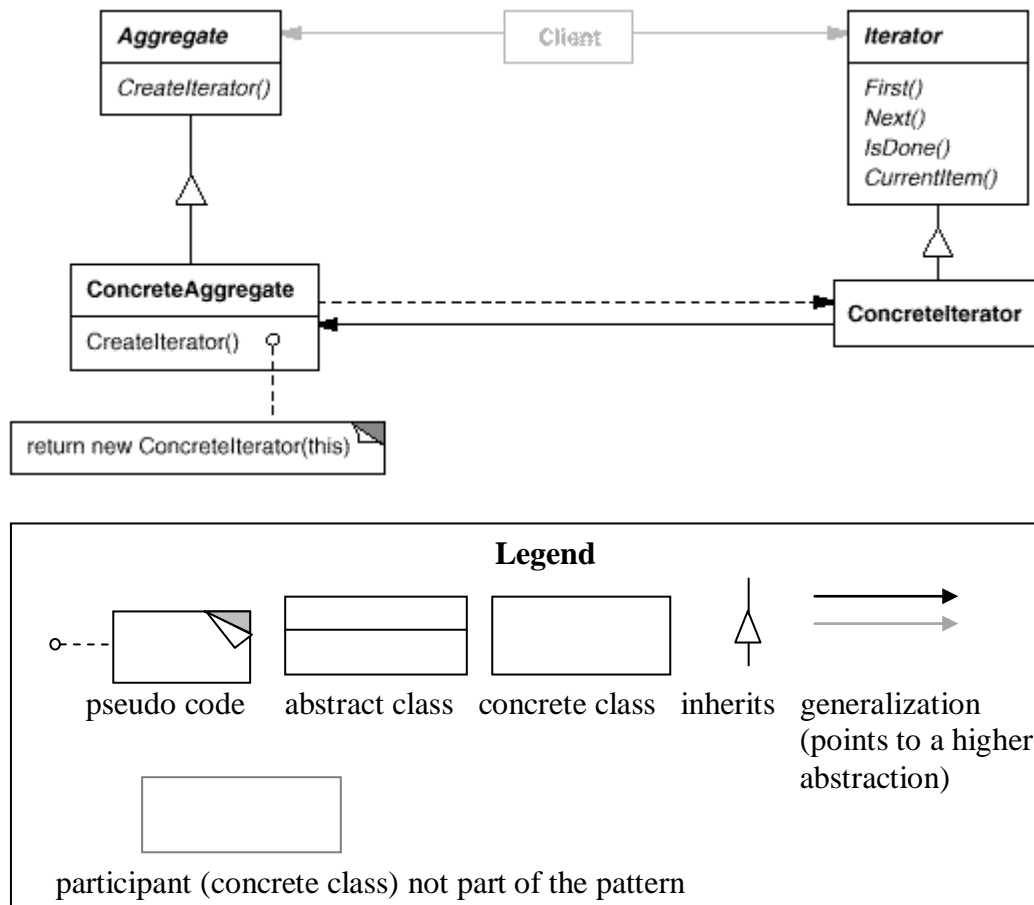
**Structure:**



Figure A-9: UML Representation of the Iterator Pattern [Gamma et al. 1995]

**Participants:**

- Iterator: Defines an interface for accessing and traversing elements.

- ConcreteIterator: Implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.

- Aggregate: Defines an interface for creating an Iterator object.

- ConcreteAggregate: Implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

**TLG Specification of the Iterator Pattern:**

Interface Aggregates
      createIterator:
End Interface

concreteAggregate :: List; ArrayList; . . . ; Array; . . . concreteAggregate-1; . . .
                  concreteAggregare-n.

Class concreteAggregate
      concreteAggregate_Iterator : create and return concreteAggregateIterator.
End Class

Interface Iterator
      First:.
      Next:.
End Interface

Class concreteAggregate_Iterator implements Iterator
      First : return first element of concreteAggregate.
      Next: return next element of concreteAggregate.
End Class

State

**Intent:** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
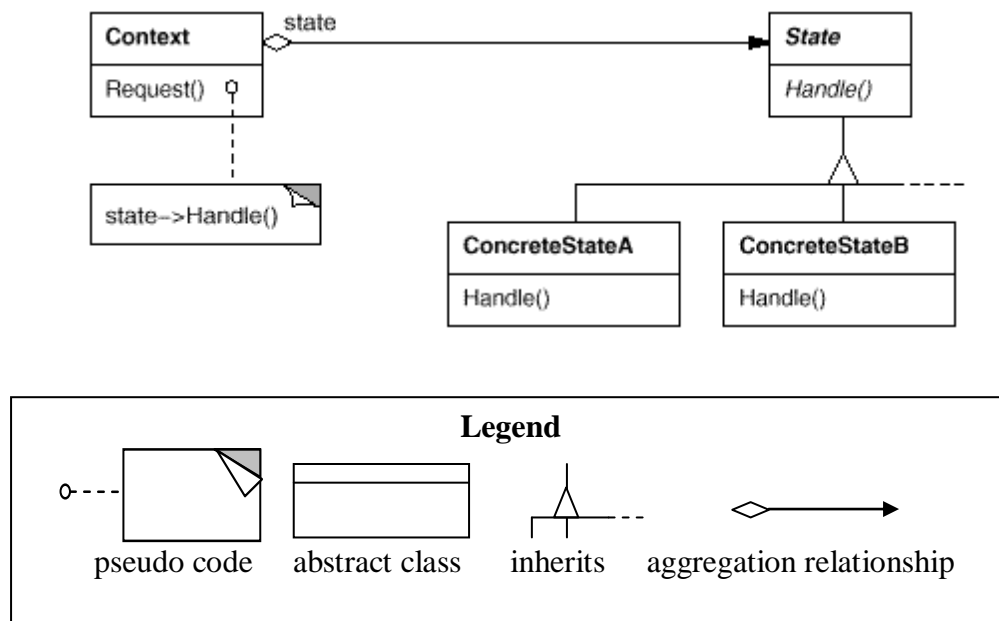
**Structure:**



Figure A-10: UML Representation of the State Pattern [Gamma et al. 1995]

**Participants:**

- Context (TCPConnection): Defines the interface of the interest to clients and maintains an instance of a ConcreteState subclass that defines the current state.

- State (TCPState): Defines an interface for encapsulating the behavior associated with a particular state of the Context.

- ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed): Each subclass implements a behavior associated with a state of the Context.

**TLG Specification of the State Pattern:**

Abstract Class State

Abstract Request:.
End Class

Class concreteState extends State
        Requests:
End Class

Class Context
        State concreteState :: concreteState-1; . . . ; concreteState-n.
        Requests : concreteState.Requests.
End Class

 Class Client
        Ops:Context.Request.
End Class

Strategy

**Intent:** Define a family of algorithms, encapsulate each one, and make them interchangeable. The strategy lets the algorithm vary independently from clients that use it.
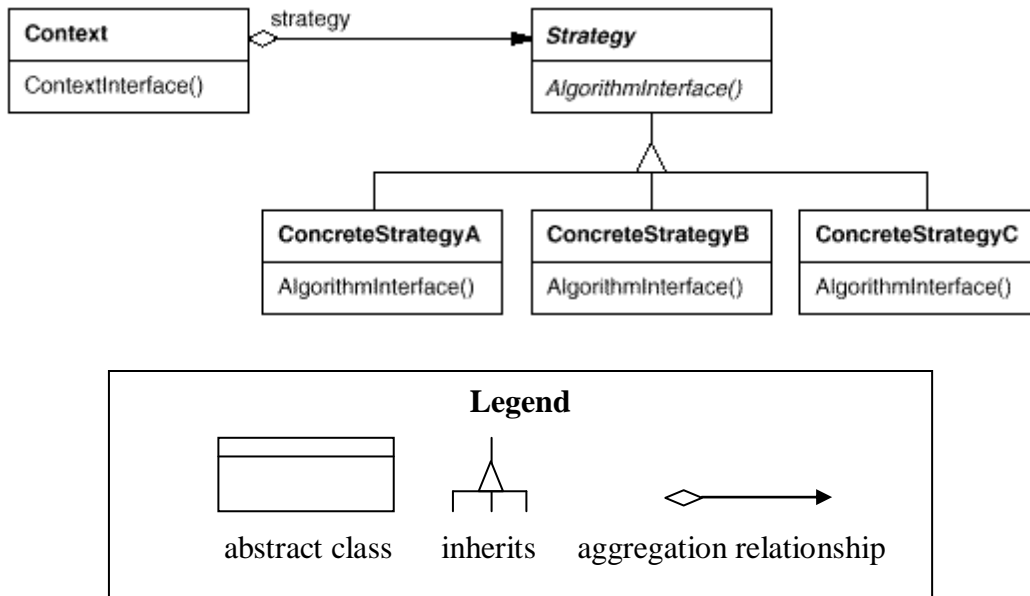
**Structure:**



Figure A-11: UML Representation of the Strategy Pattern [Gamma et al. 1995]

**Participants:**

- Strategy (Compositor): Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

- ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor): Implements the algorithm using the Strategy interface.

- Context (Composition): Is configured with a ConcreteStrategy object, maintains a reference to a Strategy object, and it may define an interface that lets Strategy access its data.

**TLG Specification of the Strategy Pattern:**

Class context
      Context :: concreteStrategy-1; . . . ; concreteStrategy-n.
      Strategy newStrategy :: Context.
      Request : newStrategy.AlgorithmInterface.
End Class

Interface Strategies
      Context AlgorithmInterface:.
End Interface

Class concreteStrategy-1 implements Strategies
      Context :: Strategy-1.
      Context AlgorithmInterface: algorithm for Strategy-1.
End Class

Class concreteStrategy-n implements Strategies
      Context :: Strategy-n.
      Context AlgorithmInterface: algorithm for Strategy-n.
End Class

Template Method

**Intent:** Define the skeleton of an algorithm in an operation, deferring some steps to the subclasses. The template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
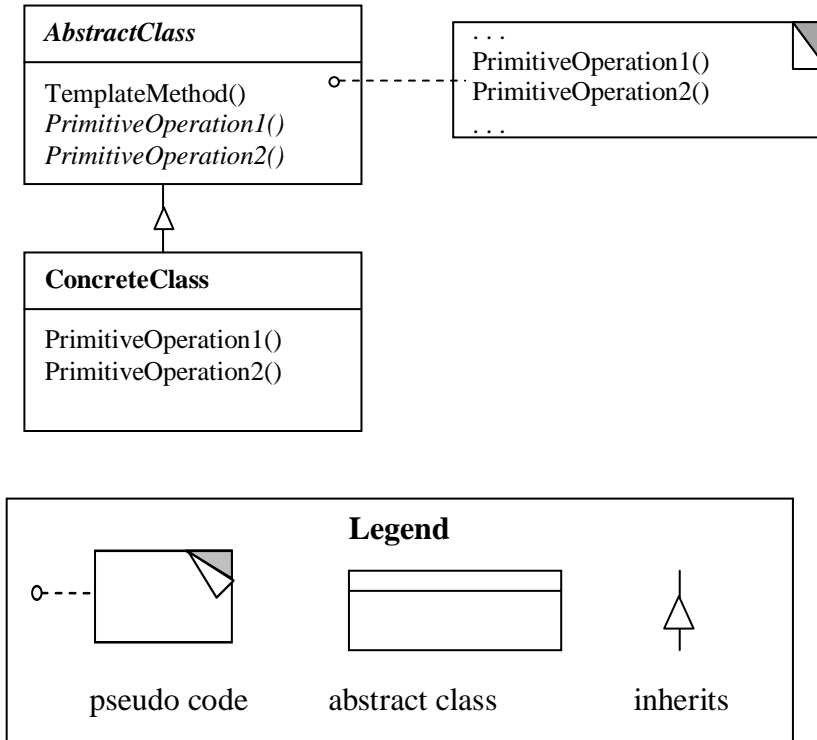
**Structure:**



Figure A-12: UML Representation of the Template Method Pattern [Gamma et al. 1995]

**Participants:**

- AbstractClass (Application): Defines abstract primitive operations that concrete subclasses define to implement the steps of an algorithm, and implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

- ConcreteClass (MyApplication): Implements the primitive operations to carry out the

  subclass-specific steps of the algorithm.

**TLG Specification of the Template Method Pattern:**

Abstract Class abstract
      PrimitiveOps :: primitiveOps-1; . . . ;primitiveOps-n.
      Abstract PrimitiveOps:.
      templateMethod : primitiveOps.
End Class

Class concrete extends abstract
      PrimitiveOps :: primitiveOps-1; . . . ;primitiveOps-n.
      PrimitiveOps: some primitive operation.
End Class

**Intent:** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
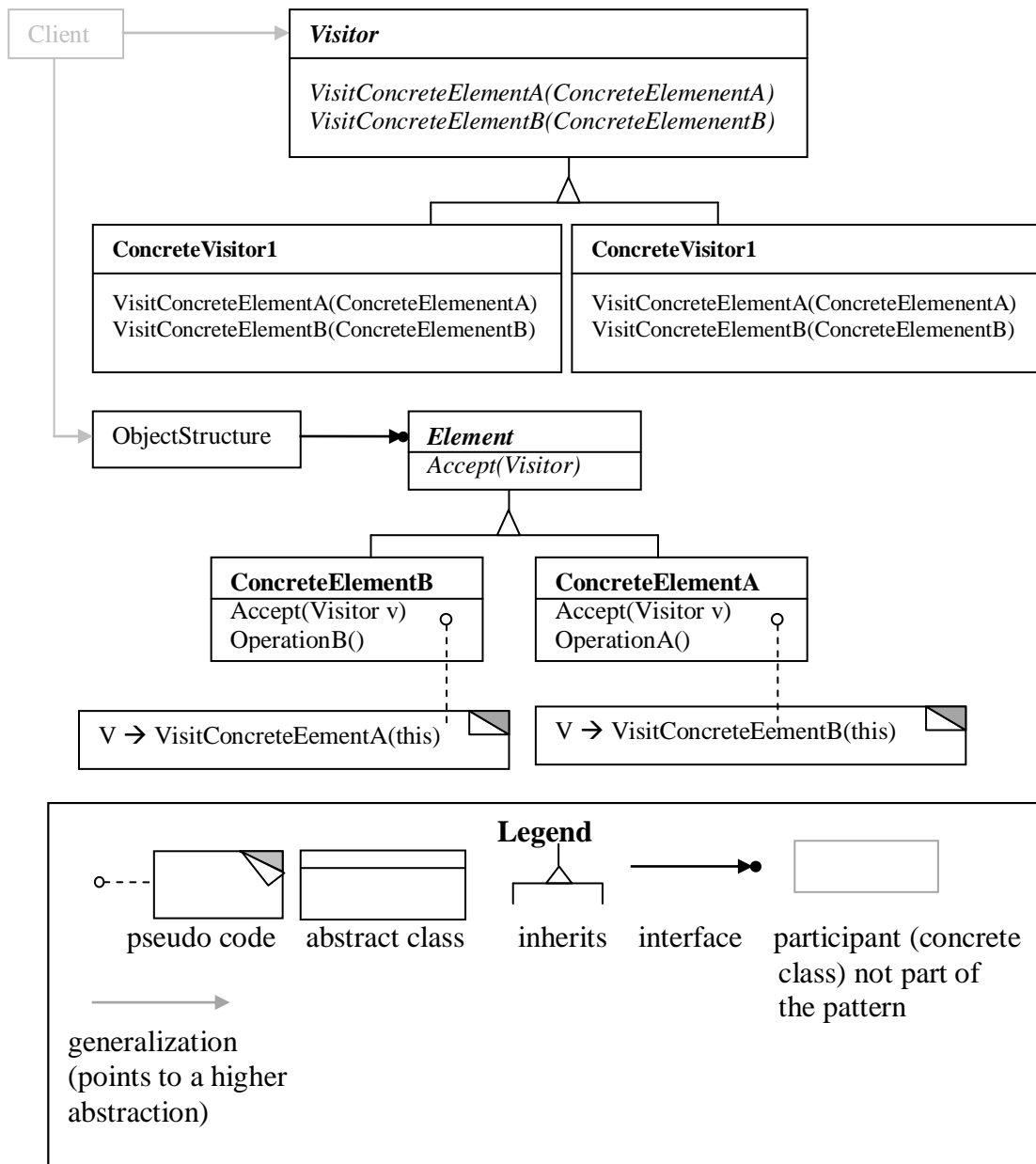
**Structure:**



Figure A-13: UML Representation of the Visitor Pattern [Gamma et al. 1995]

**Participants:**

- Visitor (NodeVisitor): Declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. The visit request lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

- ConcreteVisitor (TypeCheckingVisitor): Implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of objects in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

- Element (Node): Defines an Accept operation that takes a visitor as an argument.

- ConcreteElement (AssignmentNode,VariableRefNode): Implements an Accept operation that takes a visitor as an argument.

- ObjectStructure (Program): Can enumerate its elements, may provide a high-level interface to allow the visitor to visit its elements and it may either be a composite or a collection such as a list or a set.

**TLG Specification of the Visitor Pattern:**

```
Interface Visitor
        Visit:.
End Interface

concreteVisitors :: concreteVisitor-1; . . . ; concreteVisitor-n.

Class concreteVisitors
        Elements elements :: concreteElement-1; . . . ; concreteElement-n.
        Visit: elements.
 End Class
```

Interface Elements
  abstractVisitor::.
  abstractVisitor Accept:.
End Interface

Class concreteElements
  Visitor concreteVisitors :: concreteVisitor-1; . . . ; concreteVisitor-n.
  Accept: concreteVisitors.Visit.
 End Class

Class client
  Visitor concreteVisitors :: concreteVisitor-1; . . . ; concreteVisitor-n.
  Elements elements :: concreteElement-1; . . . ; concreteElement-n.
  Ops: elements.Accept.
End Class

# VITA

Deepa Balasundaram

Candidate for the Degree of

Master of Science/Arts

Thesis: FORMAL SPECIFICATION OF DESIGN PATTERNS:

A COMPARISON OF THREE EXISTING APPROACHES AND

PROPOSING TWO-LEVEL GRAMMARS AS A NEW APPROACH

Major Field:  Computer Science

Education: Bachelor of Science degree in Computer Science from School of Engineering and Technology, Bharathidasan University, Tiruchirapalli, Tamilnadu State, India in June 2006;  completed the requirements for the Degree of Master of Science in Computer Science at the Computer Science Department of Oklahoma State University in July 2010.

Name: Deepa Balasundaram                    Date of Degree: July 2010

Institution: Oklahoma State University          Location: Stillwater, Oklahoma

Title of Study: FORMAL SPECIFICATION OF DESIGN PATTERNS:
              A COMPARISON OF THREE EXISTING APPROACHES AND
              PROPOSING TWO-LEVEL GRAMMARS AS A NEW APPROACH

Pages in Study: 141                    Candidate for the Degree of Master of Science

Major Field: Computer Science

Patterns are Object-Oriented reusable units. The principal idea behind patterns is to capture and reuse the abstractions that have been formed by expert programmers and designers to solve problems that occur in particular contexts. These abstractions capture the valuable experiences of experts in solving problems. Although patterns are currently being used successfully, there is no general agreement among the software community as to how patterns should be formalized or represented. Various formal specification schemes have been proposed to complement the natural language description of patterns in order to alleviate the ambiguities inherent in the natural language description by rigorously reasoning about the structural and behavioral aspects of patterns. Existing formal specification languages of design patterns have generally failed to provide a standard definition, specification, or representation for patterns because there is no general agreement as to how patterns should be formalized. Also, each formal specification is generally based on a different mathematical formalism and when pattern users want to understand a pattern, first they have to understand the respective mathematical formalism.

In addition to comparing three existing formal specification schemes, the main objective of this research work was to lay the foundation for developing a formal specification scheme that could be understandable without having to delve into the details of the underlying formalism. This research work attempted to capture and represent the structural aspects of design patterns since capturing the behavioral aspects of design patterns is a semantic issue and is beyond the scope of this work. Two-Level Grammar (TLG) was used to capture and represent the structural aspects of design patterns. This study was conducted using the GoF design patterns [Gamma et al. 1995]. It has already been demonstrated that TLGs have the capability to represent the building blocks of object-oriented software systems. The primary advantage of TLGs in defining design patterns is that specifications written in TLGs are understandable due to their natural-language-like vocabulary [Edupuganty 1987] [Lee 2003] [Maluszynski 1984]. The TLG representation of the observer pattern was developed to gauge the feasibility of the proposed pattern representation scheme. TLGs could help pattern users understand the formalized version of patterns more readily compared to other formal specification methods that are difficult to understand due to their arcane mathematical notations.

ADVISOR'S APPROVAL:   Dr. M. H. Samadzadeh