DESIGN PATTERN CONTRACTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

Jason O. Hallstrom, B.S., M.A., M.S.

* * * * *

The Ohio State University

2004

Dissertation Committee:

Neelam Soundarajan, Adviser
Anish Arora
Bruce W. Weide

Approved by

[Signature]
Adviser
Department of Computer Science and Engineering
ABSTRACT

A design pattern describes a commonly recurring problem in the design of object-oriented software, a solution to that problem, and the context in which the solution can be applied. The benefits of design patterns are two-fold. First, patterns serve as guidance to the novice designer. Second, they provide an extended vocabulary for documenting software design. In the mid 1990s, the publication of several pattern catalogs – compendiums of design patterns – popularized patterns as a tool for designing object-oriented software. Today, many of these catalogs are standard references, and the patterns they contain are an important part of software practice.

Unfortunately, the descriptive format popularized by these catalogs is inherently imprecise. As a consequence, it is unclear when a pattern has been applied correctly, or what can be concluded about a system implemented using a particular pattern. This ambiguity threatens to undermine the two principal benefits associated with the use of design patterns. First, novice designers are more prone to error without a precise description of how each pattern must be applied. Second, documentation describing the patterns applied in designing a system may be misleading, as different designers can interpret pattern descriptions in subtly different ways.

To address the ambiguity issues associated with design pattern descriptions, we introduce the concept of a design pattern contract as a formalism for precisely specifying design patterns. Like all contracts, a design pattern contract consists of two
primary components: a *responsibilities* component and a *rewards* component. The responsibilities component of a pattern contract precisely characterizes the requirements that must be satisfied by the designer when applying a particular pattern. The rewards component specifies the system properties that are guaranteed to be exhibited if the contract responsibilities are indeed satisfied.

The contract formalism alone, however, is not enough to guarantee that design patterns will be applied correctly. After all, software design is a complex task. Even when guided by a precise set of requirements, designers can — and do — make mistakes. To detect such contract violations, we introduce the notion of a *contract monitor* — an executable unit of deployment used to detect runtime contract violations. During the testing phase of a system’s life cycle, the designer can instrument the system with the appropriate monitors to determine if the contracts used in its construction were applied correctly. We present two approaches to implementing these monitors. The choice of which to use depends on the requirements of the system, and the skill-set of the designer.
To my family.
ACKNOWLEDGMENTS

During the last six years at Ohio State, I have been fortunate to be involved with a number of people that have encouraged, mentored, and inspired me. First, I would like to express my sincerest gratitude to my advisor, Neelam Soundarajan. Neelam has been closely involved with every aspect of the work presented in this dissertation, and there isn’t a single chapter that doesn’t benefit from his insights. He has been an excellent advisor and friend, and I appreciate his efforts tremendously. I would also like to thank Anish Arora, Bruce Weide, and Bill Leal, who have been my unofficial co-advisors throughout the years. They have provided invaluable guidance and support, and have taken a personal interest in my success. These gents — my four advisors — are constant reminders of why I wanted to become a professor, and the relationships that I hope to build with my own students.

I am also grateful to the numerous graduate student colleagues that have shared this experience with me. In particular, I would like to thank the student members of the Reusable Software Research Group — Nigamanth Sridhar, Scott Pike, Hilary Pike, Christopher Bohn, Ben Tyler, and Michael Gibas — as well as the student members of the Dependable Distributed and Networked Systems Group — Santosh Kumar, Murat Demirbas, Hongwei Zhang, Sandip Bapat, Vinayak Naik, Vinod Kulathumani, and Prabal Dutta. These folks have all provided valuable assistance at some point or another, and have helped to make my time at OSU a period that I will look back on
with fond memories. I am especially thankful for my friendships with Nigamanth and Scott. The “team support” that they have shown throughout the years has made a tremendous difference during a number of difficult times. I will dearly miss our late night research sessions in the halls of Dreese Labs — aerobie and foosball included.

I would also like to express my gratitude to the remaining members of the Reusable Software Research Group — Bill Ogden, Paolo Bucci, Tim Long, Bob Mathis, Wayne Heym, and Joan Krone. This group has been instrumental in helping me to develop my research and presentation skills — not to mention, toughening my skin.

On the other side of the fence — outside of graduate school — I have also been blessed with a team of supporters for whom I am grateful. I would like to thank my friends from Metavante — Bob Stoll, Ruth Wollan, Bastian Knoppers, and many others — for sticking with me throughout the years, despite a schedule that made their lives difficult. I would like to especially thank Bob, who has been a dear friend and mentor, and provides one of the best examples of personal integrity that I know of. I feel very fortunate to have worked with the Willowbrook team, and I count the experience as instrumental in my academic success.

I also owe a debt of gratitude to my longtime friend, Adam Foit, whose friendship has been a source of great joy for many years. I am thankful not only for his friendship and laughter, but for his willingness to persist in his efforts to keep in contact, despite my tendency to disappear for months at a time. He has been an invaluable ally, and I want to express my most sincere appreciation.

Throughout this process, and throughout my life, I have been supported by the unwavering love and encouragement of my wonderful parents, Duane and Barbara Hallstrom. Their confidence in me has always surpassed my confidence in myself, and
as a result, I’ve been able to reach goals — humble as they might be — that once seemed unattainable. From the very beginning, that’s how it’s always been — my parents have encouraged me to excel beyond the vision that I had for myself. I can never thank them enough for all that they have done, and for the life that they have made possible for me.

I would also like to thank the rest of my family — who are too numerous to name individually — for their love and encouragement, as well as the interest that they’ve shown in my academic career. I have dedicated my dissertation work to them, because being a part of the Hallstrom family is the thing of which I am most proud.

Last, but certainly not least, I want to thank my fiancé, Kimberly Rose, for her unconditional love, laughter, and friendship. She is a reminder to me of all that is good within us, and all that I aspire to be. I could not have accomplished this task without her support, nor would I have been inclined to. Kimberly is my best friend, my love, and my life.
VITA

May 4, 1976 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Born – Palos Heights, Illinois

June 2004 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.S. Computer Science & Engineering,
The Ohio State University

August 1998 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . M.A. Economics,
Miami University

May 1998 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . B.S. Systems Analysis,
Miami University

January 2002 – present . . . . . . . . . . . . . . . . . . . . . Graduate Research Associate,
The Ohio State University

September 1998 – December 2001 . . . . . . . . . . . . . . Graduate Teaching Associate,
The Ohio State University

August 1995 – present . . . . . . . . . . . . . . . . . . . . . . Independent Systems Consultant

PUBLICATIONS

Research Publications


FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Software Methodology  Prof. Neelam Soundarajan
                     Prof. Bruce W. Weide
Software Systems      Prof. Anish Arora
Computer Graphics     Prof. Roger Crawfis
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Abstract</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>viii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 The Problem</td>
<td>3</td>
</tr>
<tr>
<td>1.3 The Approach</td>
<td>5</td>
</tr>
<tr>
<td>1.4 The Thesis</td>
<td>8</td>
</tr>
<tr>
<td>1.5 Contributions</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Organization of the Dissertation</td>
<td>9</td>
</tr>
<tr>
<td>2. Design Patterns</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>2.2 From Buildings to Software</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Pattern Examples</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1 Observer Pattern</td>
<td>17</td>
</tr>
<tr>
<td>2.3.2 Memento Pattern</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Definition Ambiguity</td>
<td>20</td>
</tr>
<tr>
<td>2.5 Chapter Summary</td>
<td>22</td>
</tr>
</tbody>
</table>
3. Design Pattern Contracts
   3.1 Chapter Overview .................................................. 24
   3.2 Responsibilities and Rewards ..................................... 24
   3.3 Role Models .......................................................... 27
   3.4 Pattern Instances .................................................... 29
   3.5 Contract Structure .................................................. 30
      3.5.1 Pattern-Level Elements ....................................... 30
      3.5.2 Role-Level Elements .......................................... 37
   3.6 Verification Overview .............................................. 44
   3.7 Contract Examples .................................................. 47
      3.7.1 Observer Pattern .............................................. 47
      3.7.2 Memento Pattern .............................................. 66
   3.8 Formalization Benefits ............................................ 80
      3.8.1 Improved Precision ........................................... 80
      3.8.2 Improved Flexibility ......................................... 82
   3.9 Chapter Summary ................................................... 83

4. Contract Monitoring Using Dynamically Generated Wrappers ......... 85
   4.1 Chapter Overview .................................................... 85
   4.2 Detecting Contract Violations ..................................... 86
   4.3 The Monitoring Approach ........................................... 88
      4.3.1 Contract Monitors ............................................ 88
      4.3.2 Role Monitors ................................................ 90
      4.3.3 The Dynamic Proxy API ..................................... 95
   4.4 Example: The Observer Pattern ................................... 97
      4.4.1 Observer Pattern Contract Monitor .......................... 99
      4.4.2 Sample Application .......................................... 112
   4.5 Discussion .......................................................... 118
   4.6 Chapter Summary ................................................... 121

5. Contract Monitoring Using AOP ........................................ 123
   5.1 Chapter Overview .................................................... 123
   5.2 Aspect-Oriented Programming with AspectJ ....................... 124
   5.3 The Monitoring Approach ........................................... 127
   5.4 Example: The Observer Pattern ................................... 131
      5.4.1 Observer Pattern Contract Monitor .......................... 131
      5.4.2 Sample Application .......................................... 140
   5.5 Discussion .......................................................... 150
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Observer Pattern</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>Memento Pattern</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Role Specifications</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Monitoring Class Structure</td>
<td>89</td>
</tr>
<tr>
<td>4.2</td>
<td>Structure of a Role Monitor</td>
<td>92</td>
</tr>
<tr>
<td>4.3</td>
<td>Structure of a Role Monitor with Mapped Assertion Checks</td>
<td>95</td>
</tr>
<tr>
<td>4.4</td>
<td>Structure of a Dynamic Proxy</td>
<td>97</td>
</tr>
<tr>
<td>4.5</td>
<td>Using the Dynamic Proxy API</td>
<td>98</td>
</tr>
</tbody>
</table>
# LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Observer Pattern: Pattern-Level Specification (1 of 2)</td>
<td>48</td>
</tr>
<tr>
<td>3.2 The Observer Pattern: Pattern-Level Specification (2 of 2)</td>
<td>49</td>
</tr>
<tr>
<td>3.3 The Observer Pattern: Role-Level <strong>Subject</strong> Specification (1 of 2)</td>
<td>56</td>
</tr>
<tr>
<td>3.4 The Observer Pattern: Role-Level <strong>Subject</strong> Specification (2 of 2)</td>
<td>57</td>
</tr>
<tr>
<td>3.5 The Observer Pattern: Role-Level Observer Specification</td>
<td>64</td>
</tr>
<tr>
<td>3.6 The Memento Pattern: Pattern-Level Specification (1 of 2)</td>
<td>67</td>
</tr>
<tr>
<td>3.7 The Memento Pattern: Pattern-Level Specification (2 of 2)</td>
<td>68</td>
</tr>
<tr>
<td>3.8 The Memento Pattern: Role-Level Originator Specification</td>
<td>75</td>
</tr>
<tr>
<td>3.9 The Memento Pattern: Role-Level Memento Specification</td>
<td>78</td>
</tr>
<tr>
<td>4.1 Observer Contract Monitor: SubjectDefs Interface</td>
<td>100</td>
</tr>
<tr>
<td>4.2 Observer Contract Monitor: ObserverDefs Interface</td>
<td>101</td>
</tr>
<tr>
<td>4.3 Observer Contract Monitor: ObserverCM Class (1 of 2)</td>
<td>103</td>
</tr>
<tr>
<td>4.4 Observer Contract Monitor: ObserverCM Class (2 of 2)</td>
<td>105</td>
</tr>
<tr>
<td>4.5 Observer Contract Monitor: ObserverIH Class (1 of 2)</td>
<td>108</td>
</tr>
<tr>
<td>4.6 Observer Contract Monitor: ObserverIH Class (2 of 2)</td>
<td>111</td>
</tr>
<tr>
<td>4.7 Observer Pattern Example: Task Interface</td>
<td>113</td>
</tr>
<tr>
<td>4.8 Observer Pattern Example: TaskDefs Class</td>
<td>114</td>
</tr>
<tr>
<td>4.9 Observer Pattern Example: ProgressBarDefs Class</td>
<td>116</td>
</tr>
<tr>
<td>4.10 Observer Pattern Example: ButtonDefs Class</td>
<td>117</td>
</tr>
<tr>
<td>5.1 Observer Contract Monitor: ObserverCM Aspect (1 of 3)</td>
<td>132</td>
</tr>
<tr>
<td>5.2 Observer Contract Monitor: ObserverCM Aspect (2 of 3)</td>
<td>136</td>
</tr>
<tr>
<td>5.3 Observer Contract Monitor: ObserverCM Aspect (3 of 3)</td>
<td>139</td>
</tr>
<tr>
<td>5.4 Observer Pattern Example: IOTask Class</td>
<td>141</td>
</tr>
<tr>
<td>5.5 Observer Pattern Example: TaskProgressBar Class</td>
<td>142</td>
</tr>
<tr>
<td>5.6 Observer Pattern Example: TaskOutputButton Class</td>
<td>142</td>
</tr>
<tr>
<td>5.7 Observer Pattern Example: IOTaskCM Aspect (1 of 2)</td>
<td>143</td>
</tr>
<tr>
<td>5.8 Observer Pattern Example: IOTaskCM Aspect (2 of 2)</td>
<td>145</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

“Clarity, clarity, surely clarity is the
Most beautiful thing in the world...”

— George Oppen.

1.1 Background

Software systems have become an integral part of the global economy, impacting almost every aspect of our daily lives. Along with the increase in stature comes an increased sense of urgency in addressing a longstanding software engineering dilemma. How do we remedy software’s chronic crisis [Gibbs 1994]? What are the tools and techniques required to produce quality software on-time and on-budget? This question lies at the heart of all software engineering research, and is the umbrella question addressed by this dissertation.

Not surprisingly, software reuse strategies form the cornerstone of most solutions. By reusing existing artifacts when developing new systems, fewer lines of code need to be written. More importantly, fewer lines of code need to be understood. While code production can be increased using automated tools, there is little that can be done to increase the intellectual capacity of the human mind. Complex concepts, if
not described in terms of more intellectually manageable concepts, will forever remain complex. When designing something as complex as software, reuse strategies are the most important tools in the practitioner’s toolbox. Unfortunately, the contents of this toolbox are in constant flux.

The constant fluctuation is driven by an unprecedented rate of growth in system size and complexity. By some accounts, the size of the average system is growing by an order of magnitude every five to ten years [Humphrey 2001, Gibbs 1994]. To keep pace, the concepts used to construct these systems must abstract over increasingly more of the code that ultimately runs on the processor.

Historically, this evolutionary process dates back to the very dawn of digital computing, when scarce resources dictated reuse via subroutines [Clements 1996]. Later, the first high level programming languages such as Fortran and ALGOL [Sebesta 2002] provided constructs for reusing common patterns of assembly language instructions. Subroutines written in these languages therefore offered larger units of reuse than those written in assembly language directly.

Still later, languages such as SIMULA, CLU, and Modula introduced the notion of a program module, a collection of subroutines operating over shared state. While the motivation for introducing program modules varies among these languages [Krogdahl 2003, Liskov 1993, Wirth 1995], these three languages helped to form the foundation for module-based programming. Continuing the evolutionary trend, module-based development provides a higher degree of reuse than that offered by simple language constructs and subroutines alone because the units of reuse are generally much larger.

Jumping ahead, object-oriented programming languages such as C++, Java, and C# currently dominate the software landscape. Consequently, “program module” has
become synonymous with “class”, the construct offered by these languages to develop program modules. In this new landscape, reuse strategies continue to evolve. Over the last decade in particular, *architectural reuse* strategies have become a focal point. These strategies increase the degree of reuse by spanning class boundaries, in effect offering even larger units of reuse. Examples in this category include object-oriented frameworks [Fayad and Schmidt 1997], container architectures [Microsystems 2001, al. 1999, Hallstrom, Leal and Arora 2003], and the focus of this dissertation, design patterns [Gamma, Helm, Johnson and Vlissides 1995, Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996, Shalloway and Trott 2001].

1.2 The Problem

Less obvious than the improvements in time and cost-to-market achieved through reuse are the inherent quality improvement benefits. Software concepts that have been realized in running systems have withstood the test of time; they are more likely to characterize good design decisions. Systems built using these concepts are therefore more likely to be of high quality. Unfortunately, their rapid evolution threatens to nullify this benefit, thereby compromising the motivation for using the concepts in the first place. The point here is that while the tools in the practitioner’s toolbox have evolved rapidly, the instructions for using these tools have not kept pace.

In addressing this problem, we focus on *design patterns* in particular, which are widely regarded as some of the most powerful reusable concepts for designing object-oriented systems. A design pattern describes a recurring problem in the design of object-oriented software, the outline of a solution to that problem, and the conditions that must be met to apply the solution successfully. Individual patterns provide
guidance on how to structure groups of classes, as well as the interactions among these classes, to implement flexible and reliable solutions.

In the mid-to-late 1990s, the software engineering community witnessed a dramatic increase in research aimed at identifying new design patterns, and recording these patterns in a standard format. The result of this research was the publication of several pattern catalogs such as [Gamma et al. 1995, Buschmann et al. 1996, Schmidt, Stal, Rohnert and Buschmann 2000], where each catalog describes a set of design patterns in narrative form. Several of these catalogs are now considered standard references, and the patterns they contain are an important part of everyday programming practice.

And while the narrative form of these pattern catalogs has proven useful, and indeed may be essential, it is also inherently imprecise. As a consequence, it is not clear when a pattern has been applied correctly, or what can be concluded about a system implemented using a particular pattern. Hence, the design guidance provided by individual patterns is muffled by the techniques used to describe them.

How can we characterize design patterns precisely, so as to fully realize their benefits? How can we reason about the behavior of systems constructed using particular patterns? How can we determine if those patterns have been applied correctly? These are the types of questions that the work presented in this dissertation will try to answer.

For variety, we say that a pattern has been applied, or that it has been implemented. The two terms are used interchangeably hereafter.
1.3 The Approach

Our approach to addressing this problem is to develop “design pattern contracts”, design pattern specifications based on the role modelling work of Reenskaug [Reenskaug 1996]. In this context, a role is defined as a view of an object with respect to a single area of concern. That is, a role is a projection of an object, such that only those aspects pertinent to a particular design concern are visible. We use these roles to model the objects that collaborate to implement particular design patterns. Our contracts formalize their structural and behavioral properties, as well as the constraints that bind their patterns of interaction. These contracts characterize not only the requirements that must be satisfied when applying each pattern, but also the system properties that are guaranteed as a result.

It is important to note, however, that the development of design pattern specifications poses an inherent risk. The hallmark of software design patterns is the flexibility that they afford to system designers; formalization efforts run the risk of compromising this flexibility [Riehle 1997]. To prevent specification rigidity, our approach is to develop parametric specifications. Each contract is parameterized by a number of “auxiliary concepts” — relations that capture points of variation across applications of the same pattern. These concepts allow us to precisely specify the design pattern properties of interest without over-constraining possible implementations. At the same time, we are able to specify constraints on these concepts, providing guidance to the designer on how particular patterns might be specialized. As we will see in Chapter 3, our approach not only avoids a compromise in flexibility, but actually improves the flexibility of certain patterns.
Contracts alone, however, are not enough to solve the problem that we’re addressing. It’s true that a pattern contract provides precise guidance on how to apply a particular pattern, but how can a designer check whether they’ve met the appropriate requirements? After all, software design is a complex task. Even when programming to specifications, designers can — and often do — make mistakes. Given this simple fact, there ought to be a way for a designer to check that the appropriate contract requirements have been met. Only then, if the contract requirements have been met, can the designer expect the system to exhibit the properties guaranteed by the contract.

Our approach to addressing this problem is to develop techniques for monitoring whether the contracts used in designing a particular system are respected at runtime. We propose two such techniques, each of which relies on a distinct set of supporting language mechanisms. Depending on the language mechanisms available, and the skill set of the designer, one approach may be preferable to the other.

The first monitoring technique requires the designer to place calls to a monitoring library at the points where verification conditions arise in the program code. In some cases, the corresponding assertions can be checked by the library directly at these points. However, when an object declares its intent to participate in a pattern, the monitoring library cannot directly determine whether the object provides the appropriate method implementations. Instead, the object must itself be monitored throughout its participation in the pattern. Hence, at the point an object declares its intent to participate in a pattern, the monitoring library dynamically generates a checking wrapper [Edwards, Shakir, Sitaraman, Weide and Hollingsworth 1998] for the object, with which it is then exchanged. The checking wrapper appears to client
objects as the original itself, but additionally checks that the appropriate behavioral requirements are indeed satisfied. Of course, the assertions to be checked vary per pattern application. Therefore, the designer is additionally required to supply objects that are used by the library to customize its behavior as appropriate to the system being monitored.

The second monitoring technique relies on aspect-oriented programming (AOP) principles [Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier and Irwin 1997, Kiczales, Hilsdale, Hugunin, Kersten, Palm and Griswold 2001]. In this context, an “aspect” modularizes code fragments that are “woven” through one or more classes, as well as the rules for how this weaving should be performed. For a particular design pattern, the monitoring code common across all instances of the pattern is modularized within a single abstract aspect. Analogous to an abstract class, the abstract aspect defers portions of its implementation to one or more subaspects. More precisely, the abstract aspect defers the portions of the monitoring code that vary among implementations of the same pattern, as well as the rules for how the monitoring code should be woven for a particular application. Hence, the abstract aspect and the subaspect together form the complete monitoring code for a particular pattern application. As we will see, this monitoring approach has a number of advantages. However, in cases where aspect-oriented programming support is unavailable, the former approach offers a suitable alternative.

Our solutions are guided by the fact that software design is in part a social process. Industry moves slowly in incorporating the results of academic research [Gibbs 1994]. Practitioners become accustomed to the tools and techniques they’ve used when developing systems, and not surprisingly, tend to resist dramatic changes to either. To
overcome this obstacle, we focus on techniques that require only modest changes to the current state of programming practice. Accessibility to practitioners is a guiding concern throughout.

1.4 The Thesis

The work presented in this dissertation defends the following two-part thesis.

- A parametric specification approach that identifies the structural and behavioral properties of program roles, as well as the interactions among these roles, provides an effective technique for precisely specifying design patterns.

- A checking approach based on the use of dynamically generated wrappers or aspect-oriented programming provides an effective technique for monitoring whether design pattern specifications are respected at runtime.

1.5 Contributions

The work presented in this dissertation makes the following principal contributions.

- **Design Pattern Contracts.** We present an approach to precisely specifying the structural and behavioral properties of software design patterns. Through the formalization of two patterns, we show that the approach makes clear the requirements that a designer must meet when applying a particular pattern, as well as the system properties that are guaranteed by virtue of applying the pattern correctly. In addition to the reliability improvements that come by way
of formalization, we show that the specifications do not compromise design pattern flexibility, but rather, elucidate opportunities for increasing their flexibility over that allowed by the original narrative characterizations.

- Monitoring of Design Pattern Contracts. We present two runtime monitoring approaches for determining whether a pattern implementation satisfies its specification. By applying these approaches to a standard design pattern, we show that each offers an effective strategy for developing pattern checking modules. More concretely, we show that each approach allows for the modular, reusable expression of the monitoring code required to check the requirements associated with individual design patterns.

1.6 Organization of the Dissertation

The organization of this dissertation is as follows. In Chapter 2, we present an overview of design patterns, including a brief discussion of their history in the object-oriented community. We explain the narrative structure used to describe design patterns, and use two examples to illustrate the ambiguities these structures present. In Chapter 3, we introduce design pattern contracts as a technique for resolving these ambiguities. Each contract precisely specifies the implementation requirements associated with a particular pattern, as well as the system properties that are guaranteed if the pattern is implemented correctly. We demonstrate the approach in the context of the pattern examples presented in Chapter 2. In Chapter 4, we introduce the notion of a contract monitor as a mechanism for determining whether the contracts used in developing a system are respected at runtime. We present an approach
to developing contract monitors based on the use of dynamically generated checking wrappers [Edwards et al. 1998]. We demonstrate the approach in the context of a system designed using one of the pattern contracts presented in Chapter 3. In Chapter 5 we present an alternative monitoring strategy based on the use of aspect-oriented programming (AOP) [Kiczales et al. 1997]. While the monitoring approach requires specialized knowledge and development tools, it improves upon many of the deficiencies associated with the wrapper-based approach. For comparison purposes, we present an AOP-based monitor applied to the same system presented in Chapter 4. In Chapter 6 we present an overview of related work, and compare our specification approach to similar techniques. Finally, we summarize our research contributions in Chapter 7 and provide pointers to future work.
CHAPTER 2

DESIGN PATTERNS

“None of us is as smart as all of us!”

— Gerald Weinberg.

2.1 Introduction

In any design intensive discipline, practitioners are confronted with recurring design problems. Each problem presents itself in many different ways, especially when designing something as malleable as software. Consequently, new designers often fail to recognize problems that they or others have encountered previously, which prevents them from applying existing solutions. The result is that each design problem is solved from scratch, and fails to benefit from the lessons learned when similar solutions were applied in the past.

Over time, however, design experience overcomes this barrier. By working on different types of design tasks, experienced designers encounter each design problem in its various forms. Solving variations of these recurring problems allows the designer to better recognize the fundamental problems being manifested. With this knowledge, new variants of the same problem are more easily recognized, and new solutions are guided by past experience. Moreover, with a better understanding of the problem’s
fundamental characteristics, the solutions become increasingly elegant and uniform. Eventually, the expert designer is able to think in terms of generalized problem-solution pairs, which form the basis for all of their design work. Indeed, as crystalized in [Buschmann et al. 1996], this is what makes the designer an expert.

Patterns seek to record the expert knowledge contained in these problem-solution pairs. Each pattern describes a generalized problem, the fundamental solution to that problem, and the context in which the solution can be applied [Alexander 1979]. While patterns have been studied in a number of domains, our focus is on patterns which capture expert knowledge in the design of object-oriented systems. More specifically, our interest is in patterns which address language independent problems associated with the design of object-oriented subsystems (i.e., groups of collaborating objects) — design patterns [Gamma et al. 1995, Coplien and Schmidt 1995, Vlissides, Coplien and Kerth 1996, Buschmann et al. 1996, Shalloway and Trott 2001].

The problems addressed by design patterns are varied. How should objects implement methods that can be undone? How should incompatible interfaces be reconciled? How can a large number of small objects be implemented efficiently? The patterns characterize these types of problems, generalized solutions to these problems, and the conditions that must be met to apply the solutions successfully. More concretely, each design pattern provides guidance on how to structure individual classes or groups of classes, as well as the interactions among those classes, to implement flexible and reliable solutions in specific contexts.

Design patterns are intended to be both prescriptive and descriptive. They are prescriptive in that they serve as guidance for novice developers. Since patterns are the product of distilled expert knowledge, applying the relevant patterns should lead
to higher quality systems with predictable behavior. They are descriptive in that they serve to better document software design. When a new developer joins a team that documents its work with respect to the patterns employed, she will be able to more quickly form a thorough understanding of how the system is structured, and why it behaves in particular ways.

2.2 From Buildings to Software

The use of patterns as a tool for capturing design experience dates back to the building architect, Christopher Alexander [Alexander 1964, Alexander, Ishikawa and Silverstein 1977, Alexander 1979, Lea 1994]. Alexander argued that traditional design methodologies are deficient in their ability to produce designs of the highest quality. This quality, which he refers to as the “quality which has no name”, is that intangible thing which cannot be characterized, but is easily recognized when seen. And while traditional techniques cannot achieve it, he argues, there are designs that undeniably possess it.

Alexander’s solution was to draw from the collective wisdom of the designers responsible for these designs, by observing recurring patterns within them. As a means of conveying this knowledge to other designers, he codified these patterns in a standard narrative format. In [Alexander et al. 1977], Alexander presents 253 patterns, each consisting of five parts: (i) a descriptive pattern name, (ii) a description of the problem, (iii) a description of a generalizable solution, (iv) the context in which the solution applies, and (v) a prototypical example. These patterns are arranged in what Alexander refers to as a “pattern language”, a structure that links the individual
patterns into a cohesive whole. In essence, the pattern language serves as a guide for
designers, so the appropriate patterns are applied at each step in the design process.

Beck and Cunningham [Beck and Cunningham 1987, Cunningham and Beck 1987, Beck 1988] were the first to adapt the ideas of Christopher Alexander to software. Their goal was to enable the design of effective user interfaces by non-programmers. To that end, they proposed a small pattern language consisting of five patterns, which could be used to guide interface design tasks. They also began the early work on more general patterns for object-oriented software. While their implementation target was the SmallTalk programming language, their patterns were relatively language independent, focusing on higher-level design concerns.

Coplien [Coplien 1992, Coplien 1998] was also interested in software patterns around the same time, with a focus on lower-level patterns for the C++ programming language. His patterns address questions such as how to efficiently implement object assignment, and how to separate implementations from interfaces. These types of patterns, characterized by their dependence on a particular programming language, are commonly referred to as “programming idioms” [Buschmann et al. 1996].

Gamma [Gamma 1991] was also a member of the patterns community, and a key player in the formative stages of design patterns’ history. As part of his Ph.D. dissertation, Gamma identified a number of recurring patterns in the design of object-oriented software. These patterns helped form the basis of the first design pattern catalog [Gamma et al. 1995], which Gamma co-authored with Helm, Johnson, and Vlissides. These authors, collectively known as the “Gang of Four”, were influenced by the work of Beck, Cunningham, and Coplien, as well as other members of the
patterns community. Their catalog is widely regarded as *the* book which sparked the popularity of design patterns, and led to the development of other pattern catalogs, such as [Buschmann et al. 1996, Schmidt et al. 2000]. Each of these catalogs is now considered a standard reference, and the design patterns they contain are an important part of everyday programming practice.

Although the catalogs vary with respect to the types of patterns they describe, they do share a structure based on Alexander’s work. First, each pattern is given a short name that reflects the purpose of the solution that it characterizes, as well as a succinct statement of that purpose. Second, the pattern description provides a characterization of the design problem it addresses, as well as a corresponding solution. The solution is described in terms of the structural and behavioral properties of the classes participating in the solution. Narrative descriptions form the bulk of the content, including discussions of the ways in which the pattern can be tailored to individual systems. This information is supplemented with UML class diagrams [Booch, Rumbaugh and Jacobson 1998, Jacobson, Booch and Rumbaugh 1999] — or similar diagrams — and code samples that represent a prototypical pattern application. Third, the pattern description provides a discussion of the consequences of applying the solution that it contains. This discussion includes notes on the solution’s advantages, as well as associated difficulties or drawbacks. Finally, the pattern description provides a list of known examples in existing software systems, as well as pointers to related patterns that might influence, or be affected by the pattern being presented.

\[2\] The chronology described in this section, as well as the determination of the key pattern originators, is based on the historical accounts provided in [Beck, Coplien, Crocker, Dominick, Meszaros, Paulisch and Vlissides 1996, Gamma et al. 1995].
This latter information induces a structure reminiscent of Alexander’s pattern language.

### 2.3 Pattern Examples

Despite the growing number of patterns that have been identified, the original design patterns published by the Gang of Four continue to be regarded as some of the most useful. Their importance to the object-oriented community is evidenced by their continued examination in the literature, their use in university curricula, and their incorporation into commercial class libraries [Microsystems 2003]. Given their importance, we consider two representative examples from [Gamma et al. 1995] to illustrate the essential characteristics of a design pattern. We refer back to these examples later when we consider the problems of how to specify and monitor patterns in the next few chapters.
2.3.1 Observer Pattern

When designing systems in the object paradigm, it is common for a group of objects to depend on a single other object. Often, the dependent objects must remain consistent with the object on which they depend. The canonical example of this situation is in the context of designing graphical user interfaces. Consider, for example, an object that encapsulates the behavior required to complete some long running task. Next, consider two user interface objects used to signal the user of the task’s progress. The first object renders a linear bar graph, the length of which is proportional to the task’s completion percentage. The second object is a standard button used to initiate the display of the task’s output. This button remains disabled (i.e., unresponsive to clicks) until the task completes. The state of each user interface object must remain consistent with the state of the task object, which continues to change as the task is executing. What is the appropriate technique for achieving this behavior? This is precisely the question addressed by the Observer pattern.

The pattern describes an approach to keeping a set of Observer objects consistent with the state of some Subject object. The basic idea is for the subject to maintain a logical set of interested observers, and to alert those observers whenever its state is modified. When alerted of a change, each observer updates its state to become consistent with the new state of the subject.

3See Chapter 6 for a detailed discussion of the design patterns literature.

4We use names beginning with uppercase letters, such as Subject and Observer, to refer to pattern participant types. We use the corresponding lowercase names to refer to objects participating as those types. We also use names starting with uppercase letters for patterns, classes, and methods; member variables will have names starting with lowercase letters. In some cases the name of a pattern is also used for one of the participant types in that pattern, as in the case of the Observer participant type in the Observer pattern. In such cases, the context will make clear whether we are talking about the participant type or the pattern.
A prototypical example of the pattern is illustrated in Figure 2.2. An observer, or some object acting on behalf of the observer, declares the observer’s interest in the subject by calling the subject’s Attach() method. Similarly, when an observer is no longer interested in remaining consistent with the subject, it can be detached using the subject’s Detach() method. Whenever the subject’s state is modified, the Notify() method is used to alert each interested observer of the change. In turn, the Notify() method calls Update() on each interested observer, prompting the observer to become consistent with the new state of the subject. As each observer relies on the state of the subject in a different way, the implementations of Update() vary from one observer to the next. Again, the diagrams and code fragments represent a prototypical realization of the pattern, and do not capture all of the allowable variations. While some variations are discussed in the original characterization, it is ultimately up to the designer to generalize the solution as appropriate to the systems in which it is used.
2.3.2 Memento Pattern

Another common design scenario is for an object to require its state to be stored, so that at some later point in the object’s lifetime, this state can be used to restore the object to its previous value. Consider, for example, a drawing application in which Shape objects can be resized or moved. If an object is resized incorrectly, the user might like to undo the operation, requiring the value of the Shape object to be restored. If resize operations are negotiated against a grid, simply storing the resize distance is insufficient to restore the object to its previous value. Resizing the object in the opposite direction of the stored displacement might result in the object being resized incorrectly. If the interfaces of the objects involved provide insufficient flexibility to reverse an undoable operation, the state of the Shape object must be saved. However, some of the state components encapsulated by the object might not be accessible through the Shape’s interface. The object might, for example, track the number of resize operations initiated on the object by the user. One possibility for solving this problem is to expose the private state of the Shape object so that each state component can be stored individually. However, thirty years of research in information hiding dictates that this practice undermines scalable design principles [Parnas and Siewiorek 1975]. Breaking encapsulation exposes both the object and its clients to interference side-effects that thwart local reasoning. What is the appropriate technique for exposing the state of an object so that the object can later be restored? How can this be done without violating encapsulation? These are the questions addressed by the Memento pattern.

The intent of the Memento pattern is to externalize some portion of an object’s abstract state without exposing the object’s concrete representation, so that the value
of the object can be restored at a later point. The pattern identifies three participant types: Originator, Memento, and Caretaker. The originator is the object whose state is to be externalized, and provides an interface for creating memento objects that encapsulate this externalized state. Mementos are maintained by caretakers, which can use the mementos to restore the state of the originator.

A prototypical example of the pattern is illustrated in Figure 2.2. When the originator requires checkpointing, say due to a pending operation that might later be undone, the CreateMemento() method is used to create a memento that encapsulates the state of the originator at the time of the call. Should the originator need to be restored, this memento can be passed to the SetMemento() method. The method restores the state of the originator to that encapsulated by the memento passed as argument. While the memento provides an interface for accessing its state, this interface must only be used by the originator that constructed it; caretakers must never access the state of a memento. The motivation for including this constraint is to prevent memento objects from revealing state components that might violate the encapsulation of the originator. Again, this is a prototypical example which must generalized, and then tailored to the needs of the application in which the pattern is to be applied.

2.4 Definition Ambiguity

While narrative descriptions of the type summarized above are useful, and may even be essential, they leave a number of important questions unanswered. Consequently, different designers may have different notions of what constitutes a particular

---

5A more suitable name for this method would be something along the lines of RestoreOrig(). However, we use SetMemento() to remain consistent with the original pattern description.
pattern’s implementation. This result has serious implications for the utility of design patterns, as ambiguity in their descriptions undermines their prescriptive and descriptive benefits. Their prescriptive benefits are compromised since novice designers do not have a precise description of how a particular pattern should be applied. They might therefore apply the pattern incorrectly. Similarly, the descriptive benefits of patterns are eroded since different designers documenting their software with respect to the patterns employed might mean subtly different things. The resulting documentation is therefore less effective than it might otherwise be.

Consider some of the questions that arise in the context of the Observer pattern. Recall that when the state of the subject is modified, the subject’s Notify() method must be called to update each of the attached observers. But what does it mean for the state of the subject to be modified? Do we mean that each time any bit within the subject changes, all attached observers should be updated? Surely there are changes that can be made to the subject which do not count, at least as far as the observers are concerned, as having modified the subject’s state. Specifically, observers are unlikely to be concerned with subject changes which do not affect their consistency with the subject. But now there’s another question; what do we mean by consistency? A definition of consistency that calls for the state of each observer to be identical to the state of the subject is far too restrictive to be generally applicable. Indeed, under this definition, the observers would have to be updated each time even a single bit within the subject changes. So what do we mean — precisely?

Similar questions arise if we consider the Memento pattern. Recall that only the originator is allowed to access the interface provided by the memento for accessing its state. Consider a caretaker that stores many mementos, and buffers them to disk. As
part of its buffering strategy, it might need to record the number of times a particular memento has been buffered. The designer could of course store this information in an associative lookup (e.g., a hash-table), but the more natural approach in the object paradigm is store this information within the memento. If the caretaker were to update this information, would the collaboration still be considered an instance of the Memento pattern? According to the narrative description, the answer is no, but this seems like an unreasonable restriction.

Consider another interesting question. Suppose at time $t_2$, the state of an originator is restored using a memento created at an earlier time $t_1$. What should happen to the mementos created between $t_1$ and $t_2$? Should these mementos be considered defunct, since the originator was restored to an earlier point than the one at which these mementos were created?

Each of these questions points to the same fundamental issue. What precisely are the requirements that must be met to apply a particular design pattern correctly? Perhaps more interesting to the system designer, if the pattern is applied correctly, what can the designer conclude by virtue of having applied it? As we will see in the next chapter, these are the questions that pattern contracts address.

### 2.5 Chapter Summary

A pattern describes a recurring design problem, the core of a solution to that problem, and the context in which the solution can be applied [Alexander 1979]. The focus of this dissertation is on design patterns for object-oriented software. These patterns are intended not only to serve as guidance to novice developers, but also to provide a new vocabulary for documenting software design. With knowledge of
the patterns employed in a particular system, a new designer is more quickly able to understand how the system is structured, and why it behaves in particular ways.

Of the numerous pattern catalogs that have been published, the Gang of Four’s catalog [Gamma et al. 1995] continues to be one of the most influential. Their narrative structure, which consists primarily of textual descriptions supplemented with prototypical code examples, has become the defacto standard for describing design patterns. While these descriptions have proven valuable, they are inherently ambiguous and incomplete. As a result, the exact requirements that must be met when applying a particular design pattern are unclear, as are the exact consequences of having applied it.
CHAPTER 3

DESIGN PATTERN CONTRACTS

“The art of reasoning becomes of first importance.”
— Thomas Jefferson.

3.1 Chapter Overview

In this chapter, we introduce the notion of a design pattern contract. Each contract precisely characterizes the requirements that must be satisfied when applying a particular pattern, as well as the system properties that are guaranteed if the pattern is applied correctly. After presenting the contract formalism, we demonstrate the approach in the context of two examples. We conclude with a discussion of how the resulting contracts simultaneously improve upon the precision and flexibility offered by existing narrative characterizations.

3.2 Responsibilities and Rewards

Regardless of the domain of discourse, a contract describes a binding agreement between two parties. Although individual contracts vary in their precision, they do share a common structure. Every contract outlines a set of responsibilities that need
to be satisfied, and the \textit{rewards} that will be received for having satisfied those responsibilities. Consider, for example, the case of a homeowner contracting a plumber to fix a leaky faucet. When the two parties enter into the contract, the plumber agrees to satisfy his responsibility of fixing the leak, and expects to be rewarded with payment by the homeowner. The contract is a device for describing this agreement.

A design pattern contract is a similar instrument. In this context, the contract is a binding agreement between the designer and the pattern being applied. And like any contract, it consists of two principal components: a responsibilities component and a rewards component. The responsibilities component specifies the conditions that the designer must satisfy when applying the pattern that the contract characterizes. These conditions include requirements on the structures of participant classes, the behaviors of methods in these classes, and the interactions among the corresponding instances. The rewards component specifies the properties that the resulting system is guaranteed to exhibit by virtue of having applied the pattern correctly. That is, by virtue of applying the pattern as dictated by the contract, the designer is rewarded with the ability to conclude properties about the systems in which the pattern has been applied.

It is worth mentioning at this point that the “responsibilities and rewards” approach to specifying design patterns bears similarity to standard design-by-contract techniques [Meyer 1987, Meyer 1997, Liskov and Wing 1994, Leavens and Weihl 1995]. In design-by-contract, the methods of each class are precisely characterized by a contract that binds the caller and the callee. More concretely, each method is described in terms of a “pre-condition” and a “post-condition”. The pre-condition associated with a method characterizes the requirements that must be satisfied before the method can
be called. Similarly, the post-condition describes the properties that are guaranteed by the method upon termination — assuming the pre-condition was satisfied when the call was initiated. When calling a method, it is the caller’s responsibility to ensure that the corresponding pre-condition is satisfied. In return, the caller is guaranteed that when control returns, the corresponding post-condition will be satisfied. In our formalization of patterns, the design pattern contract specifies the set of requirements that the system designer wishing to use the pattern must meet. In return, the designer is assured that the reward associated with the pattern will be satisfied.

It’s not surprising that our approach to specifying design patterns is similar to existing approaches to specifying the behavior of classes. After all, design patterns are a natural extension of classes. Whereas classes focus on how individual objects behave, patterns are concerned with interactions among groups of objects. And while it might be possible to encapsulate the objects participating in a particular design pattern within a larger class — in which case we would again be in the realm of design-by-contract — this defeats their intent. A pattern is intended to describe a behavioral and structural slice through an application design. This cross-cutting structure is part of what makes the pattern a good solution in the eyes of the object-oriented community. We want to think of the objects participating in the pattern as individual entities and consider the interactions among them. Design-by-contract allows us to precisely specify individual classes; design pattern contracts allow us to precisely specify how groups of objects collaborate to implement individual patterns.
3.3 Role Models

While design patterns describe the state and behavior to be provided by individual objects participating in a solution, they do not offer complete characterizations of the participating objects. Each pattern characterizes only those aspects relevant to a single area of concern — namely, the implementation of the pattern in question. When the pattern is actually applied, the objects participating in the pattern are likely to be involved in areas of concern outside of the pattern’s implementation. The state and behavior provided by the objects to support these additional areas of concern are left unspecified in the pattern’s description. In formalizing design patterns, the same principle must apply. The contract must specify (and therefore constrain) only those aspects of each object that are required to support the pattern in which it participates.

To allow for this type of partial specification, our design pattern contracts build on Reenskaug’s role modelling work [Reenskaug 1996]. Under his definition, a “role” is a projection of an object with respect to a particular area of concern. All aspects of the object (e.g., state components, method behaviors) that are irrelevant to that concern are invisible under this projection. Or stated another way, a role is a type of filter, through which the designer can focus on issues specific to a single area of concern. Although Reenskaug’s work is not focused on formal specification, the notion of a role is central to our approach.

Concretely, a design pattern contract associates a role with every type of participant involved in the pattern. Included in the contract is a “role-level specification” corresponding to each of the identified roles. The role-level specification formalizes the responsibilities associated with the objects that wish to play the corresponding
role in the pattern. As these are *role-level* specifications, and not object (i.e., class) specifications, they specify only those responsibilities required to support the object’s part in the pattern. The conceptual difference between a class specification and a role specification is illustrated in Figure 3.1.

While each role focuses only on the responsibilities associated with implementing a particular pattern, the contract often needs to refer to additional state and behavior. The contract might, for example, specify that a change in *any* state component of an object — including those components that support areas of concern outside of the pattern’s implementation — must result in some sequence of method calls. Or perhaps the contract specifies that *all* object methods satisfy some property. The state components provided by an object playing a particular role that are in addition to the state components required by the pattern are collectively referred to as the “application-level state” of the object. Similarly, the methods provided by an object that are in addition to the methods required by the pattern are collectively referred to as the object’s “application-level methods”. As we will see when we turn our attention
to the structure of a design pattern contract, the formalism allows the specifier to impose constraints on application-level state and behavior without compromising the flexibility that design patterns afford.

Finally, it is important to note that the lifetime of a role may be different than the lifetime of the object that plays it. For example, when an object is first constructed, it need not immediately participate in any patterns. At some later point, however, it might be necessary for the object to assume a role in some existing pattern. This object is said to have “enrolled” in a particular role, and is referred to as the “player” of that role. When an object isn’t participating in any patterns, there is no reason for the object to respect a role-level specification of any kind. It is only after an object has enrolled in a particular pattern that the object becomes bound by the appropriate role-level specification.

3.4 Pattern Instances

Implicit in our discussion to this point is the fact that during a system’s execution, there may be multiple instances of the same design pattern present. Consider, for example, the Observer pattern discussed in Chapter 2. At any instant during the run of a particular system, it is possible for two observers ob1 and ob2 to be observing some subject object su1. At the same time, there might be two other observer objects ob3 and ob4 observing some other subject object su2. In this situation, ob1, ob2, and su1 form a group of objects collaborating according to the Observer pattern, while ob3, ob4, and su2 form a second (entirely unrelated) group collaborating according to the same pattern. At this point in the system’s execution, there are two instances
of the Observer pattern present. Therefore, when an object enrolls to play a role in a pattern, it is actually enrolling to play that role in some instance of the pattern.

3.5 Contract Structure

We now turn our attention to the structure of a design pattern contract. Each contract consists of a “pattern-level” portion, and a “role-level” portion that contains the role-level specifications corresponding to each role. The pattern-level portion of the contract characterizes properties such as pattern instantiation conditions, player enrollment conditions, and the reward for applying the pattern correctly. As discussed previously, each role-level specification describes the responsibilities associated with the objects that play a particular role. We describe the principal elements of the pattern-level and role-level portions of the contract in the subsections that follow. To better understand these elements, and their relations to the contract structure as a whole, the reader may want to periodically refer to the example contract — corresponding to the Observer pattern — presented in Section 3.7.1.

3.5.1 Pattern-Level Elements

- Participating Roles. The pattern-level portion of a design pattern contract begins by enumerating the roles that participate in an instance of the pattern. Each role is identified by a name, which serves as a forward reference to the corresponding role-level specification. The contract further specifies the number of objects that are allowed to play each role in an instance of the pattern.
Later, in specifying the responsibilities associated with individual roles, role names will serve an additional purpose — they will be used to refer to programming language types. More specifically, a role name may be used to refer to the *type* of object that plays that role in an instance of the pattern. In this sense, every contract is parameterized by a set of type parameters — one parameter corresponding to each role that participates in the corresponding pattern. As we will see when we consider the examples presented in Sections 3.7.1 and 3.7.2, these parameters are used in specifying the state components and method behaviors that must be provided by the objects playing each role. At the point of pattern instantiation (a concept to be discussed shortly), the designer must supply the type of object used to realize each role in that instance of the pattern. These actual parameters are then used in instantiating the individual role-level specifications as appropriate to the pattern instance.

- **State Requirements.** The contract next specifies the state components that must be provided by objects wishing to play each role. Again, these state components do not fully characterize the state of any player object. Instead, the contract specifies only those state components required to support the player’s role-level behavior — as prescribed by the corresponding role-level specification. An object enrolling to play a particular role is likely to provide additional state components that support its interactions in other areas of concern. While the contract may later need to refer to these application-level components, it will not characterize the elements explicitly. Instead, the contract refers to these components collectively as “ασ”, denoting the *application-level* state of the object.
Additionally, the contract might specify state components required by the pattern that are not associated with any of the constituent roles. These requirements are, in a sense, requirements on the *static* state of the pattern. For example, a particular pattern might require aggregate data to be stored about all of the objects enrolled in an instance of the pattern. In many cases, there is a distinguished role that must be played by exactly one object throughout the pattern’s lifetime. In these patterns, it is possible to associate the static state requirements with the distinguished role — although this may not be the most natural solution. In other patterns, however, there are no distinguished roles. Consider, for example, a pattern in which there are two roles, and multiple players may enroll in each. In this case, the static state requirements are not easily associated with a particular role, but rather, with the pattern itself. The contract corresponding to the pattern would therefore specify pattern-level state requirements.

Note that the specified state requirements are requirements on the *abstract* state that must be provided, and do not necessarily correspond to field requirements. At the point an object enrolls in a particular role, the designer must supply suitable abstraction functions [Hoare 1972] for each of the state components required by the role-level specification. Each abstraction function precisely describes how the state of the player object represents a particular role-level state component. Similarly, at the point a pattern is instantiated, the designer must supply abstraction functions for each of the pattern-level state components. As
we will soon discuss, these functions are used in showing that each of the individual player methods meets the requirements imposed on it by the role-level method specification.

- **Auxiliary Concepts.** Recall that one of the principal factors contributing to the success of design patterns is the flexibility that they afford to system designers. Each pattern presents a generalizable solution that can be tailored as appropriate to a wide range of design scenarios. In specifying design patterns, care must be taken to ensure that this flexibility is not compromised. Such a compromise would limit the applicability of design patterns, thereby reducing their utility to system designers. On the other hand, misapplication of a pattern is not likely to result in high-quality software.

Formalization efforts should therefore not be precluded. After all, there are aspects of every pattern that recur among instances of that pattern. Indeed, if this were not the case, there wouldn’t be any *pattern* to speak of! These common aspects are the aspects that should be defined explicitly — they are the core of the solution characterized by the pattern. However, explicit definitions for those aspects which vary must *not* be provided. Instead, the designer should be allowed to supply tailored definitions for those aspects on a per application basis.

To allow for precision without compromising flexibility, each pattern contract is parameterized by a set of “auxiliary concepts”. Each concept corresponds to an aspect that varies across instances of the corresponding pattern. Concretely,
the concepts are represented as relations over the states of the participating objects. These relations may involve the state elements required by the pattern, or more frequently, the *application-level* state of the corresponding players. These concepts are then used in specifying the behavior associated with the role-level methods, as well as in specifying the reward for having applied the pattern correctly.

When the system designer chooses to apply a particular contract, she supplies concept definitions appropriate to the system in which the pattern is being applied. Depending on the contract, a definition may be associated with the pattern as a whole, or with a particular player object. In the former case, the designer is required to supply an appropriate definition at the point of pattern instantiation (a concept to be discussed shortly). In the latter case, the designer is required to supply an appropriate definition at the point the player object enrolls in the pattern. When reasoning about the behavior of the player object, the designer appeals to the corresponding role-level specification instantiated with the concept definitions appropriate to that object.

The contract may additionally specify constraints on the allowable concept definitions. Since these concepts are used in specifying role-level behaviors, and since these roles are required to collaborate in particular ways, the allowable definitions must often be constrained. When actual definitions are ultimately supplied, it is the designer’s responsibility to show that the definitions satisfy the appropriate constraints.

Note that in order to define a relation over the *application-level* state of a player, there must be a way to precisely characterize that state. Therefore, at the point
when a player object enrolls in a pattern, the designer must supply a state model corresponding to the application-level state of that player. The state model supplied must be consistent with the model over which the auxiliary concepts are defined. Further, the designer must supply an abstraction function that precisely characterizes how the model is represented by the player’s actual state. Since role-level method behavior is specified in terms of auxiliary concepts defined over application-level state, the state model and the associated abstraction function are used in showing that each of the individual player methods satisfies its role-level responsibilities.

- **Instantiation Requirements.** The contract next specifies the conditions that must be satisfied when creating an instance of the pattern. There are two types of conditions that can be imposed. First, the contract may require that one or more roles in the pattern be filled at the point of pattern instantiation. Such conditions are typically imposed when an instance of the pattern requires particular roles to be filled throughout the pattern’s lifetime. Second, the contract will require the designer to supply definitions for pattern-level auxiliary concepts.

Note that a contract does not explicitly specify the conditions that must be met to destroy a design pattern instance. The destruction conditions are implicit when considering the contract in toto. After an instance of the pattern has been created, any violation of the contract destroys the instance in violation. After all, once the objects participating in a particular pattern instance stop behaving according to the pattern requirements, there is no pattern instance to
speak of. In a sense, the pattern has gone out of scope at this point. Once this occurs, the pattern cannot be restored — a new instance must be created.

- **Enrollment Requirements.** When an object enrolls to play a particular role, at the point of pattern instantiation or after, the contract will require the designer to supply appropriate definitions for the auxiliary concepts associated with that role. These role-level concept definitions (along with the object types corresponding to each role, the player’s application-level state model, and any pattern-level concept definitions) are used to instantiate the corresponding role-level specification. In combination with the abstraction functions supplied at this point (as well as any abstraction functions supplied at the point of pattern instantiation), the designer must then show that the behavior provided by the object satisfies its role-level specification.

The contract may, however, impose additional enrollment conditions. For example, contracts typically require that an enrolling object not already be enrolled in an instance of the same pattern. Less frequently, the contract requires certain relations to hold among the enrolled player objects and the new object being enrolled. These requirements are imposed to ensure that new objects enrolling in the pattern do not violate the set of valid pattern states. More concretely, these requirements are imposed to ensure that the player objects continue to satisfy the pattern invariant (a concept to be discussed shortly).

- **“Disenrollment” Requirements.** When an object enrolls in a particular role, it may be required to play that role throughout the lifetime of the pattern instance. In other cases, enrollment is a temporary notion. An object might
participate in a pattern instance for some period of time, and then absolve itself of role-level responsibilities by disenrolling from that instance. Just as the contract specifies the conditions that must be met when an object enrolls in a particular role, it also specifies the conditions that must be met when an object disenrolls. If these conditions are met, the reward guaranteed by the contract continues to hold. If, however, an object disenrolls without meeting the specified conditions, the corresponding pattern instance is destroyed.

- **Pattern Invariant.** The reward for applying a pattern correctly is the ability to reason about the properties of the system in which the pattern has been applied. To capture this reward, the pattern contract introduces the notion of a “pattern invariant”. A pattern invariant is a predicate, defined in terms of auxiliary concepts, that relates the states of the objects participating in an instance of the pattern. The relation is guaranteed to hold whenever control is outside of the objects participating in that instance — assuming the contract responsibilities have been met.

3.5.2 Role-Level Elements

- **Initial Conditions.** Each role specification begins by specifying the initial state conditions that must hold when an object enrolls to play the corresponding role. These conditions are similar to the enrollment conditions specified in the pattern-level portion of the contract in that they ensure that new objects enrolling in the pattern do not violate the set of valid pattern states. The conditions specified at this level, however, are constrained to a more local view of the player playing the role. That is, the enrollment conditions specified at the
pattern-level have a global view of all pattern participants, and can therefore span the state of those participants. The conditions specified at the role-level are limited to a local view of the role, and can therefore only impose conditions on the state associated with the individual players playing that role. Note, however, that the internal state of a player object is likely to include references to other objects that participate in the pattern. These referenced player objects are considered to be a part of the player object’s internal state. The initial conditions might therefore require certain relations to hold between the state of the player object and the other players that it references.

Note that the initial conditions imposed at this level could alternatively be expressed as part of the enrollment clause specified in the pattern-level portion of the contract. The ability to specify role-level initial conditions is provided as a specification convenience.

• **Role Invariant.** The role specification next identifies the invariant that must be respected by an object wishing to play the role. The invariant, like the initial conditions, imposes constraints only on the local state of the player object. The invariant is the role-level analogue of a class invariant; it specifies properties that must hold at the start and end of every method invocation — including both role-level and application-level methods. And unlike the pattern invariant which describes the reward for having applied the pattern correctly, the role invariant describes a responsibility — it tells the designer how the pattern must be applied.
• **Role-Level Method Requirements.** The role specification next characterizes the methods that must be provided by enrolling objects to support their role in the pattern. While the specification assigns a name to each required method, the actual name of the corresponding player method need not be identical. At the point of enrollment, the designer must supply a mapping that specifies the name of the player method corresponding to each role-level method. The conformance of the player object to its role-level specification is then shown under this mapping.

With respect to structural properties, the specification identifies the *minimal* set of parameters that must appear in the parameter list of each role-level method, as well as the method’s return type. Additional parameters used to support the player object’s other areas of concern are left unspecified. At the point of enrollment, the designer must supply for each method, a mapping that specifies the name of the player parameter corresponding to each parameter in the role-level method specification. The proof of conformance to the role-level specification must additionally take this mapping into account.

In specifying behaviors, each role-level method is characterized by a pre-condition, a post-condition, and a preservation constraint. The pre-condition and post-condition associated with a method have the same meaning as in standard design-by-contract. The preservation constraint identifies those state components that must not be affected by the method invocation. The construct is provided only as a specification convenience, since the equivalent requirements could alternatively be expressed as part of each method’s post-condition.
The pre-condition and post-condition associated with each method will often be expressed entirely in terms of the individual state components and auxiliary concepts. In general, however, a more elaborate mechanism is required to capture method behavior. This has to do with the fact that in some situations, the pattern dictates not that a particular method change the state to meet some specified condition, but that during its execution it make a specific sequence of calls to other methods. To specify such requirements, we make use of a “call sequence” \( \text{(i.e., trace [Buchi and Weck 1999, Soundarajan and Fridella 2000])} \), which records the calls made during a system’s execution. We assume that a new call sequence is associated with every method invocation, and that during the method’s execution, all method calls are automatically recorded in the sequence. Each sequence element is modelled as a tuple consisting of eight fields: (i) the name of the method invoked, (ii) the value\(^6\) of the caller before the invocation, (iii) the value of the caller after the invocation, (iv) the value of the callee before the invocation, (v) the value of the callee after the invocation, (vi) the value of the argument list before the invocation, (vii) the value of the argument list after the invocation, and (viii) the value (if any) returned by the method. Requirements may then be imposed on this trace as part of each method’s post-condition\(^7\).

\(^6\)The \textit{value} of an object refers to both its identity and abstract mathematical value. In the case of a player object, its abstract mathematical value includes the individual abstract values corresponding to its role-level state components, as well as the abstract value of its application-level state.

\(^7\)In the pattern-level portion of a contract, we may additionally need to refer to the “object call sequence” associated with a particular object. This call sequence records information about the methods executed on the object throughout its lifetime. A contract might, for example, require that a certain sequence of method calls occur before an object be allowed to enroll in a particular role. We will see an example of such a requirement when we turn our attention to the example presented in Section 3.7.2.
The behavior specified in the post-condition of each role-level method does not necessarily characterize the *exact* behavior required of the corresponding player method. In general, player methods — even those used to satisfy role-level requirements — provide functionality in support of additional areas of concern. To prevent a compromise in flexibility, the specification identifies only the “minimum behavior” required of each role-level method. Player methods are free to provide additional functionality, so long as they meet the minimum behavioral requirements. More precisely, when an object wishes to enroll in a particular role, the designer must show — given the appropriate type parameters, application-level state models, abstraction functions, concept definitions, and mappings — that the post-condition of each player method implies the post-condition of the corresponding role-level method.\footnote{Note that state components which are not explicitly constrained in the post-condition of a role-level method may be modified arbitrarily by the corresponding player method. This is identical to the convention used in other specification approaches such as [Edwards, Heym, Long, Sitaraman and Weide 1994, Sitaraman and Weide 1994].}

In line with behavioral subtyping [Liskov and Wing 1994] requirements, the post-condition of a role-level method always specifies the weakest condition that must be satisfied at the method’s termination. Pre-conditions introduce an interesting wrinkle. In general, the pre-condition of a role-level method can always be *weakened*\footnote{This notion is again in line with behavioral subtyping requirements.} by the actual player implementation. After all, there is no reason for a role-level specification to prevent a player method from servicing invocations that it could otherwise handle correctly. In other cases, however, it might be appropriate for the player method to *strengthen* the role-level pre-condition. That is, a player method might require additional conditions to be
satisfied, beyond those characterized by the role-level pre-condition. For ex-
ample, a player method might impose conditions on the additional state com-
ponents provided by the player in support of areas of concern outside of the 
pattern’s implementation.

However, unlike the case of weakening, not all pre-conditions can be strength-
ened. Consider, for example, a call sequence requirement that requires a partic-
ular sequence of calls be made to methods defined in the role-level specification.
In this case, the associated pre-conditions must specify the strongest condi-
tions that can be required. If this were not the case, the pre-conditions could 
be strengthened by the corresponding player implementations, making the re-
quired sequence of calls illegal. As a result, it would be impossible for the player 
to satisfy the role-level post-condition that required the sequence of calls.

To make the designer’s responsibilities clear, the specification syntax allows pre-
conditions to be tagged as can_strengthen, indicating that the player method can 
impose a stronger pre-condition. When showing that the role-level method 
responsibilities have been satisfied, the designer must take these tags into ac-
count. We will see examples of tagged and un-tagged pre-conditions when we 
turn our attention to the examples presented in Section 3.7.

A Note on Call Sequence Requirements. Expressing the appropriate 
assertions directly over a call sequence provides a general approach to capturing 
the sequence of calls that must be made during a particular method’s execution.

\[^{10}\]Note that in some cases, a method’s pre-condition may be irrelevant to the pattern’s correctness. In such cases, specifying true as the pre-condition, and tagging the pre-condition as can_strengthen, provides the appropriate flexibility. While specifying false has the same effect, the former provides a more natural specification approach.
As we will soon see, however, expressing even simple conditions can result in unwieldy specifications. More complex conditions can result in deeply nested quantifiers — evidence of an unmanageable intellectual load. Fortunately, there are at least two techniques for making these conditions more manageable.

First, the specifier can introduce a level of abstraction by providing appropriate functions over the call sequence. These functions can then be used in specifying the call sequence requirements, thereby making the resulting specification more manageable. We will see an example of this approach when we consider the contract described in Section 3.7.1.

Second, the specifier can introduce appropriate “auxiliary variables” as part of the contract specification. These variables represent “thought state”, and do not represent actual components that must be provided as part of a pattern’s implementation. Instead, the variables are used by the contract to record information about the call sequence elements as part of the post-condition of individual methods. That is, post-conditions may include assignment clauses that assign call sequence information to the appropriate auxiliary variables. The call sequence conditions — particularly those associated with other methods — can then be simplified by expressing constraints over this recorded state. We will see an example of this approach when we turn to the contract monitoring code discussed in Chapters 4 and 5.

- **Application-Level Method Requirements.** Recall that a player object is likely to provide state beyond what is required to support its role in the pattern. In the same way, the object is likely to provide additional methods. These
application-level methods are not used to support the pattern, but rather, to support the other areas of concern in which the object participates. Still, the role-level specification may need to impose constraints on these methods since patterns often require application-level methods to respect particular properties. For example, a pattern might require that only role-level methods be allowed to modify a particular role-level state component.

To specify the responsibilities associated with application-level methods, the approach is identical to that used in specifying role-level method responsibilities. Specifically, the specification identifies the pre-condition, post-condition, and preservation constraint that must be respected by all application-level methods. When an object wishes to enroll in a particular role, the designer is required to show that all of the player’s application-level methods satisfy the specified conditions.

### 3.6 Verification Overview

The verification conditions that must be satisfied when showing that a particular implementation correctly realizes its contract materialize at three points: (i) at the point when a pattern is instantiated, (ii) at the point when an object enrolls in a particular role, and (iii) at the point when an object disenrolls from its role. Before summarizing these conditions, it is useful to summarize the information that must be provided by the designer at these points. First, at the point of pattern instantiation, the designer is required to supply (i.i) the object types corresponding to the roles identified by the contract, (i.ii) the abstraction functions characterizing how the pattern-level state components are represented, and (i.iii) the definitions of all
pattern-level auxiliary concepts. At the point of object enrollment, the designer must supply (ii.i) the player’s application-level state model, (ii.ii) the abstraction function characterizing how the player’s application-level state model is represented, (ii.iii) the abstraction functions characterizing how the role-level state components are represented, (ii.iv) the mapping that specifies the player-level method corresponding to each role-level method, (ii.v) the mapping that specifies for each role-level method, the player-level parameter corresponding to each role-level parameter, and (ii.vi) the definitions of all role-level auxiliary concepts. Note that in many cases, the designer will be required to supply one or more player objects at the point of pattern instantiation. In such cases, the designer must provide items i.i—i.iii, as well as items ii.i—ii.vi; items ii.i—ii.vi must be provided per enrolling object. The designer is not required to supply any additional information at the point an object disenrolls from its role.

We now turn our attention to the verification conditions that must be dispatched by the designer. At the point of pattern instantiation, the designer must show that all required player objects have been provided, and that the supplied pattern-level concept definitions respect any constraints that have been imposed on them. The conditions that must be satisfied at the point of player enrollment — the timing of which may coincide with pattern instantiation — are more complex. First, as in the previous case, the designer must show that the role-level concept definitions respect the appropriate constraints. Second, the designer must show (using items i.ii-i.iii, ii.i-ii.iii, and ii.vi) that the player object satisfies its role-level invariant, role-level initial value constraint, and any additional constraints imposed by the enrollment clause.
Finally, the designer must show (using items i.i—i.iii, and ii.i—ii.vi) that each player-level method satisfies the responsibilities imposed on it by the role-level specification. The conditions that must be satisfied when a player disenrolls are typically the most modest. In short, the designer must show (using items i.ii-i.iii, ii.i-ii.iii, and ii.vi) that any “disenrollment” constraints imposed on the object have been met.

Note, however, that none of the points at which verification conditions must be dispatched appear explicitly in the underlying system code. Unlike a class and its methods, design patterns exist only in the eyes of the designer. Consequently, if given only the code for a system, it is generally impossible to determine where a pattern is instantiated, or where an object enrolls or disenrolls from a pattern instance. Therefore, the correctness of a particular pattern’s implementation cannot be verified.

To overcome this problem, designers might be required to introduce suitable “pattern-code” — perhaps expressed in the form of comments in the underlying programming language — at the point where a pattern is created, and at the points where an object enrolls or disenrolls from a particular pattern instance. Verification responsibilities would materialize at these points, requiring a conscientious designer to verify that the appropriate responsibilities have been met. This verification might be formal, semi-formal, or informal depending on the tastes of the designer. Alternately, or in addition, the designer might employ techniques for monitoring whether the appropriate design pattern contracts are respected at run-time. We will consider these ideas in more detail in Chapters 4 and 5.
3.7 Contract Examples

To illustrate our specification approach, we now present two example contracts corresponding to the design patterns summarized in Chapter 2. The first contract formalizes the responsibilities and rewards associated with the Observer pattern. The second contract formalizes the Memento pattern.

3.7.1 Observer Pattern

The design pattern contract corresponding to the Observer pattern is split into five separate listings. It should be noted, however, that the decision to divide the contract among separate listings was purely a presentational choice. The listings together comprise a single contract, and should not be considered as individual specifications. After all, the whole point of a design pattern is to consider the interactions among a group of objects — not to consider the objects in isolation.

Listings 3.1 and 3.2 present the pattern-level portion of the contract. Listings 3.3 and 3.4 present the role-level specification corresponding to the Subject role. Finally, Listing 3.5 presents the role-level specification corresponding to the Observer role. The individual contract elements are discussed in the subsections following each listing.

Explanation of Listings 3.1 and 3.2

- Lines 3—5: Corresponding to the participant types described in Chapter 2 there are two roles associated with the Observer pattern: the Subject role

11Following a similar convention to that used in Chapter 2, we use names beginning with uppercase letters, such as Subject and Observer, to refer to roles. We use the corresponding lowercase names to refer to objects playing those roles. We also use names starting with uppercase letters for patterns and methods; state components will have names starting with lowercase letters. In some cases the name of a pattern is also used for one of the roles in that pattern, as in the case of the Observer role in the Observer pattern. In such cases, the context will make clear whether we are talking about the role or the pattern.
Listing 3.1: The Observer Pattern: Pattern-Level Specification (1 of 2)
enrolling_as Subject:
    requires:
      false
    defines:
      null

disenrolling_as Subject:
    requires:
      false

enrolling_as Observer:
    requires:
      (player \notin \text{enrolled} \_as (Subject))
      \land (player \notin \text{enrolled} \_as (Observer))
    defines:
      \text{Consistent}(Subject.\alpha \sigma, Observer.\alpha \sigma)

disenrolling_as Observer:
    requires:
      player \notin Subject._observers

\textbf{invariant}:
  \forall ob \in Subject._observers :
  : \text{Consistent}_{ob}(Subject.\alpha \sigma, ob.\alpha \sigma)

Listing 3.2: The Observer Pattern: Pattern-Level Specification (2 of 2)
and the **Observer** role. The bracket notation following each identifier indicates the number of objects that may enroll to play that role in any instance of the pattern. Therefore, the specification indicates that in any instance of the Observer pattern, only one object may be enrolled to play the **Subject** role, and any number of objects may be enrolled to play the **Observer** role. This matches our intuition; an instance of the pattern consists of multiple **observers** observing a single **subject**.

- **Lines 7—10** Recall that the **subject** must be aware of the attached **observers**, so that the appropriate **observers** are notified when the state of the **subject** changes. Therefore, an object wishing to play the role of **Subject** is required to supply a state component corresponding to the set of attached **observers**. This state component, referred to as `observers`, stores references\(^{12}\) to the player objects that have enrolled to play the role of **Observer** and have been attached to the **subject** object. Recall that this is a requirement on the abstract state of the player enrolled in the **Subject** role. The **Set** type identified by the contract refers to a mathematically defined type that will be realized differently by individual player objects.

Similarly, recall that an **observer** must bring itself into a state that is consistent with the current state of the **subject** whenever its **Update()** method is invoked. Consequently, there must be a way for the **observer** to interrogate the state of the **subject**, so that it can determine the value of the state to become consistent.

\(^{12}\)To remain consistent with commercial languages such as Java and C#, we assume a programming model in which every variable stores an object reference. Hence, we do not introduce additional syntax for identifying reference variables, since those are the only types of variables supported in our model.
with. The contract therefore requires objects wishing to play the role of Observer to provide a state component that maintains a reference to the subject to which they are attached. Note that this state component, referred to as \_subject, is replicated — one instance corresponding to each observer.

In this pattern, there are no state components corresponding to the pattern as a whole, hence this part is specified to be null.

Once again, these state components do not fully characterize the state of the objects that participate in an instance of the pattern. Consider the Subject role for instance. Certainly the object that eventually plays this role will provide state components other than .observers. Indeed, if it didn’t, there would be nothing for the observers to observe!

**Lines 12–27** Given two values for the application-level state of a subject, what does it mean for the second state to have been modified from the first? Since the precise meaning varies from one instance of the pattern to another, Modified is listed as an auxiliary concept defined over the application-level state of the subject object. At the point where this concept is used in Listing 3.4, the definition (to be) supplied by the designer (at the point of pattern instantiation) is used to determine whether the value of the subject object at the end of a method invocation is modified from the value of the object at the start of the invocation. There is one definition of this concept per subject, and therefore one definition per pattern instance.
Similarly, the notion of what it means for the state of an observer to be consistent with the state of the subject depends on the pattern instance in question. Consequently, Consistent is listed as an auxiliary concept defined over the application-level state of a subject and an observer. At the point where this concept is used in Listing 3.5, the definition supplied by the designer is used to determine whether the observer methods (e.g. Update()) leave the state of the observer in a consistent state. And since the notion of consistency might vary from one observer to another — even within instances of the same pattern — definitions are supplied per observer.

Next, the contract specifies constraints on the concept definitions that can be supplied by the designer. The first three clauses of the outermost conjunct require that the negation of the Modified relation be reflexive, symmetric, and transitive, respectively. That is, this portion of the constraint requires the negation of the Modified relation to be an equivalence relation.

The next clause imposes a more complex constraint. To understand why this constraint is required, suppose that we have two application-level subject states, \( su.a\sigma_1 \) and \( su.a\sigma_2 \), which are identical according to the definition of Modified. Suppose further that there is an application-level observer state \( ob.a\sigma \) that is consistent with \( su.a\sigma_1 \), but inconsistent with \( su.a\sigma_2 \) according to the definition of Consistent. If one of the subject methods were to change the application-level state of the subject from \( su.a\sigma_1 \) to \( su.a\sigma_2 \), the subject’s Notify() method need not be called since the subject’s state hasn’t changed according to the definition of Modified. But according to the definition of Consistent, the observer is now inconsistent with the new state of the subject! If the contract were to allow

52
such definitions, it would allow implementations that undermine the pattern’s intent. The specified constraint prevents this problem from arising. For any pair of definitions — of Modified and Consistent — if two subject states are considered unmodified, each state must be consistent with the same set of observer states.

- **Lines 29–37** To create an instance of the Observer pattern, the contract requires that the designer supply an object to play the role of Subject. Along with this object, the designer must supply an appropriate definition for the Modified concept. Aside from the requirement that the object satisfy its role-level responsibilities detailed in Listings 3.3 and 3.4, there are no other conditions imposed on the enrolling object. Similarly, there are no auxiliary concepts associated with the pattern as a whole, so the defines block associated with the pattern instantiation block is specified as null.

- **Lines 39–43** Recall that in any instance of the Observer pattern, exactly one object must play the role of Subject. Therefore, after the pattern instance has been created, if a designer wishes to enroll another object in the role of Subject, they are required to satisfy the impossible condition of false. That is, the contract forbids the enrollment of subjects once the pattern instance has been created — ensuring only one subject per pattern instance.

- **Lines 45–47** Similarly, for a subject to disenroll from an instance of the pattern without destroying the pattern instance, it must satisfy the impossible condition false. This essentially prevents the subject object from disenrolling once an instance of the pattern has been instantiated. Again, this matches our intuition — without a subject, there is nothing to observe. Hence, if the subject
chooses to disenroll, the instance in question will be destroyed since the contract will have been breached.

- **Lines 49—54** The conditions for enrolling as an Observer once an instance of the pattern has been created are more liberal. The contract states that the object to be enrolled must not already be enrolled as either a Subject or an Observer in the current pattern instance\(^\text{13}\)\(^\text{14}\). Further, when an object enrolls to play this role, the designer must supply an appropriate definition for the auxiliary concept Consistent. This definition will be used only when reasoning about the behavior of this particular observer.

Notice that in this formalization of the Observer pattern, the enrollment and “disenrollment” of an observer object is independent of the calls made to Attach() and Detach(). Hence, the view implied by this specification is one in which observer objects periodically lose their interest in the state of the subject, but do not stop participating in the logical pattern instance. An alternative view is to merge observer enrollment with calls to the Attach() method, and to merge “disenrollment” with calls to the Detach() method. In that case, an observer object would enroll in the pattern when it was attached to the subject, and would disenroll when it was detached. The implementation responsibilities are identical under both views, as are the rewards for applying the pattern correctly.

For simplicity, we choose to present the former view. We will see an example

\(^{13}\)In general, player objects are likely to be restricted from playing multiple roles in the same pattern instance. Rather than making a fixed implicit assumption, however, we choose to specify this condition explicitly, since some patterns might require objects to participate in multiple roles.

\(^{14}\)It may be worth noting that some systems do use a version of the pattern where an object playing the Subject role also enrolls as an Observer in the same instance — for example, “private events” in the .NET Framework. An alternative specification is required to capture these systems.
of enrollment based on call sequence conditions when we turn our attention to
the Memento pattern contract presented in Section 3.7.2.

- **Lines 56—58** Unlike the subject object, observer objects may disenroll without
destroying the pattern instance in which they participate. The disenrolling
object must, however, satisfy the specified condition — namely, the disenrolling
object must not be contained in the _observers set. That is, when an observer
disenrolls, it must not be attached to the subject. This is to ensure that all
of the objects contained in the _observers set behave according to the Observer
role-level specification.\(^\text{15}\)

- **Lines 60—62** The intent of the Observer pattern is to keep a group of ob-
server objects consistent with the state of some subject object. The reward for
applying the pattern is therefore a consistency guarantee, which is captured by
the pattern invariant. The invariant asserts that for each observer in the set
_observers, the observer’s state will be consistent with that of the object playing
the role of Subject. The definition of consistency is per observer, as defined by
the designer at the point of observer enrollment.

**Explanation of Listings 3.3 and 3.4**

- **Lines 3—4** The initial condition imposed by the Subject specification requires
that at the point of enrollment, the _observers state component be empty. This
matches our intuition since the pattern cannot be instantiated until the subject
\(^\text{15}\)Recall that when an object disenrolls from its role, it is no longer obliged to respect its role-level
specification.
role_specification
Subject {
  initially:
  _observers = \Phi

  invariant:
  true

  methods:
  void Attach(Observer ob):
    requires [can_strengthen]:
      ob \notin _observers
    preserves:
      \alpha\sigma
    ensures:
      _observers = _observers@pre \cup \{ob\}
    ensures_call_sequence:
      \exists k: (cs[k].callee = ob)
      \land (cs[k].method = Update)

  void Detach(Observer ob):
    requires [can_strengthen]:
      ob \in _observers
    preserves:
      \alpha\sigma
    ensures:
      _observers = _observers@pre - \{ob\}
    ensures_call_sequence:
      true

  void Notify():
    requires:
      true
    preserves:
      _observers, \alpha\sigma
    ensures:
      true
    ensures_call_sequence:
      \forall ob \in _observers :
      \exists k: (cs[k].callee = ob)
      \land (cs[k].method = Update)

Listing 3.3: The Observer Pattern: Role-Level Subject Specification (1 of 2)
object has enrolled. Hence, there can’t be any objects enrolled as Observer, which means there can’t be any attached observers.

• **Lines [6–7]** There is no invariant associated with this role; it is trivially specified as true.

• **Lines [10–19]** Attach(), as discussed previously, is the method used to declare the interest of an observer in the state of the subject. The pre-condition of this method requires that the interested observer, ob, not already be attached. Moreover, since the type of ob is Observer — a role identified in this contract — the object passed as argument must already be enrolled in this role. That is, we implicitly assume (ob ∈ enrolled_as(Observer)). Recall that the actual type corresponding to this role will be supplied by the designer at the point of pattern instantiation.
The method’s post-condition specifies requirements on the role-level state of the subject, as well as on the method’s call sequence. First, upon termination, the method is required to have added the attaching observer to the set of interested observers. Second, the method is required during its execution to place a call to ob’s Update() method. The reason for this requirement is to ensure that the newly-attached observer — which might be inconsistent — is brought into a state that is consistent with the state of the subject.

Finally, notice that the preservation constraint requires that the method not modify the subject’s application-level state. The reason for this condition is to ensure that when control returns from Attach(), ob — along with the other observers — is consistent with the current state of the subject.

- Lines 21–29 Detach() is specified similarly. The pre-condition requires that ob is, in fact, an attached observer. Upon termination, the post-condition requires that this observer be removed from the set of interested observers. In this case, there is no need for the subject to call ob’s Update() method — hence, this requirement has been omitted. Note, however, that the omission of this requirement does not prevent the subject from making such a call.

- Lines 31–41 Recall that Notify() is the method provided by the subject to alert attached observers that its state has changed. The condition imposed on the call sequence therefore requires that Notify() call the Update() method of every attached observer. Moreover, the method is required to preserve the application-level state of the subject. Similar to the condition imposed on the

---

16If it were desirable to require such a call, the call sequence requirement imposed on the Attach() method would also be imposed here.
Attach() method, this requirement is included to ensure that at the method’s conclusion, the attached observers are consistent with the current state of the subject.

- **Lines 43—57** The others block specifies the responsibilities associated with application-level methods. That is, the specified requirements must be met by all methods other than the player-level methods corresponding to Attach(), Detach(), and Notify(). The preservation constraint requires that these application-level methods preserve the set of attached observers — otherwise the protocol implemented by Attach() and Detach() would be compromised. Further, the condition imposed on the call sequence requires that if any of these methods modifies — according to the definition of Modified provided at the point of subject enrollment — the application-level state of the subject, the call sequence must include a call to Notify(). Moreover, the state of the subject at the end of the application-level method invocation must be unmodified — again according to the definition of Modified — from the value of the subject at the point the call to Notify() was made. And finally, there must be no further calls to Notify(), as such calls could bring one or more observers into a state that is inconsistent with the final state of the subject. When we examine the monitoring techniques discussed in Chapters 4 and 5, we will consider an alternative approach to capturing these call sequence requirements based on the use of auxiliary variables.

**Correcting a Subtle Specification Error.** A more careful examination of Listings 3.3 and 3.4 reveals a subtle defect in the role-level specification. The problem has to do with the fact that when preservation constraints are imposed on individual state
components, those constraints must only be satisfied when the method in question terminates. So, for example, if a particular method is required to preserve the state of some component, an implementation of that method is free to modify the component during the method’s execution. The implementation can satisfy the preservation requirement by restoring the modified state component to its pre-conditional value.

To understand why this presents a problem in our specification, begin by noting that \texttt{Attach()}, \texttt{Detach()}, and \texttt{Notify()} are each required to preserve the application-level state of the \texttt{subject}. The preservation constraint is imposed to prevent these methods from bringing any attached \texttt{observers} into states that would be inconsistent with the final state of the \texttt{subject}. Application-level methods are not required to satisfy such a constraint. However, if they fail to preserve the application-level state of the \texttt{subject}, they are required to place an appropriate call to \texttt{Notify()}. Again, the motivation is to ensure consistency between the \texttt{subject} and each attached \texttt{observer}.

The problem arises when we consider method implementations that take advantage of the subtle flexibility provided by the definition of “preserves”. Consider, for example, an implementation of \texttt{Attach()} that simply adds the \texttt{observer} passed as argument to the set of attached \texttt{observers}, and then returns. Now consider adding a code fragment at the end of \texttt{Attach()} that sets the state of the \texttt{subject} to some temporary value, invokes the \texttt{Notify()} method, and then restores the state of the \texttt{subject} to its previous value. In such a case, the method satisfies its role-level preservation requirement, but may leave one or more attached \texttt{observers} in a state that is inconsistent with the state of the \texttt{subject}. Similar problems arise when considering malicious implementations of \texttt{Detach()} and \texttt{Notify()}.
Before turning our attention to a solution, consider the call sequence requirement imposed on application-level methods. In particular, consider an application-level method that includes the malicious code fragment described above. In such a case, since the post-conditional value of the subject would be unmodified from its pre-conditional value, the call sequence requirement would be trivially satisfied. Again, this could result in one or more observers being left inconsistent with the final state of the subject.

One approach to solving this problem is to introduce the notion of “invariant preservation”. The basic idea is to introduce a preservation requirement that must be respected throughout the execution of a method. So, for example, we could require that Attach(), Detach(), and Notify() invariantly preserve the application-level state of the subject. This would certainly solve the problem we’ve been considering, since the application-level state of the subject could never be modified during the execution of these methods. However, this constraint would be overly-restrictive, preventing implementations that required temporary modifications to the subject’s state. Moreover, this approach would be unsuitable for specifying the behavior of application-level methods. After all, if these methods were not allowed to modify the application-level state of the subject, the subject’s application-level state could never change — which would of course undermine the purpose of applying the pattern. The requirement of invariant preservation must therefore be relaxed.

In considering how to relax this requirement, note that temporary changes to the subject’s state should be allowed, so long as the state of the subject is restored before the next call to an observer method. That is, the application-level state of the subject must be preserved only with respect to each observer. State changes that cannot be
detected by the attached observers should not be disallowed by the specification. To capture this requirement, we can impose appropriate conditions on the call sequence. Specifically, we can impose a condition that requires the state of the subject to be identical to its post-conditional value whenever any observer method is executed.

Even this requirement, however, is too restrictive. During the execution of a subject method, there is no reason to require that the attached observers remain consistent with the state of the subject throughout the call. The important condition is that at the method’s conclusion, every attached observer must be consistent. Therefore, the above preservation requirement can be further relaxed. Concretely, we can impose a condition that requires the post-conditional value of the subject to be identical to its value at the point of the last call to each observer. Further, we need not require that the state of the subject be identical to its post-conditional value — it is sufficient to require that it be unmodified according to the definition of Modified supplied at the point of subject enrollment.

Although this requirement can be expressed by quantifying over the call sequence directly, the specification is more intellectually manageable if we introduce a function that abstracts over this quantification. To that end, we define the following (partial) function, expressed over the identity of the subject, the application-level state of the subject, the identity of an observer, and the call sequence associated with the invocation of a subject method.

\[
\text{StateAtFinalCall}(\text{Subject}, \text{Subject.} \alpha \sigma, \text{Observer}, \text{call-sequence}) : \text{Subject.} \alpha \sigma
\]

\[\equiv \]

\[\text{17Note that if the application-level state of the subject is required to be preserved, the pre-conditional and post-conditional values would be identical.}\]
(\text{StateAtFinalCall}(\text{sub}, \alpha_\text{sub}, \text{obs}, \text{cs}) = \text{cs}[k].\text{caller.} \alpha_\text{sub})
\iff ((\text{cs}[k].\text{caller} = \text{sub})
\land (\exists \ j : (j \geq k) \land (\text{cs}[j].\text{callee} = \text{obs}))
\land (\forall \ l : k < l < j : \((\text{cs}[l].\text{caller} \neq \text{sub}) \land (\text{cs}[l].\text{callee} \neq \text{sub})))
\land (\forall \ m : m > j : (\text{cs}[m].\text{callee} \neq \text{obs})))
\land
(\text{StateAtFinalCall}(\text{sub}, \alpha_\text{sub}, \text{obs}, \text{cs}) = \text{cs}[k].\text{callee.} \alpha_\text{sub})
\iff ((\text{cs}[k].\text{callee} = \text{sub})
\land (\exists \ j : (j > k) \land (\text{cs}[j].\text{callee} = \text{obs}))
\land (\forall \ l : k < l < j : (\text{cs}[l].\text{caller} \neq \text{sub}) \land (\text{cs}[l].\text{callee} \neq \text{sub})))
\land (\forall \ m : m > j : (\text{cs}[m].\text{callee} \neq \text{obs})))
\land
(\text{StateAtFinalCall}(\text{sub}, \alpha_\text{sub}, \text{obs}, \text{cs}) = \text{sub.} \alpha_\text{sub})
\iff (\neg \exists \ k : (\text{cs}[k].\text{callee} = \text{obs}))

For a particular observer, the function yields the application-level state of the subject at the point the observer was last called. If the call sequence doesn’t contain a call to the observer in question, the function yields the application-level state value passed as argument. With this function, we can now precisely characterize the relaxed preservation constraint with the following clause.

\forall \text{ob} \in \text{Observers : } \neg \text{Modified}(\text{StateAtFinalCall}(\text{this}, \alpha_\text{this}, \text{ob}, \text{cs}), \alpha_\text{this})

To correct the error present in Listings 3.3 and 3.4, this clause needs to be added conjunctively to the call sequence constraints imposed on each of the subject’s methods — including the application-level methods.
role_specification
Observer {
  initially:
  _subject = Subject::player
  
  invariant:
  true
  
  methods:
  void Update():
    requires:
      true
    preserves:
      _subject
    ensures:
      Consistent(_subject.ασ, this.ασ)
    ensures_call_sequence:
      true
  
  others:
  requires [can_strengthen]:
    true
  preserves
  _subject
  ensures
  Consistent(_subject.ασ, this.ασ)
  ensures_call_sequence:
    true
}
Explanation of Listing 3.5

• Lines 3–4 The initial condition requires that at the point of enrollment, the object wishing to play the role of Observer have a reference to the player object enrolled in the Subject role\(^{18}\). As every method is required to preserve this reference, the subject field references the enrolled subject throughout the observer’s lifetime.

• Lines 6–7 There is no invariant associated with this role; it is trivially specified as true.

• Lines 10–18 The post-condition of Update() requires that the method bring the observer into a state that is consistent — according to the definition of Consistent provided when the observer enrolled — with the current state of the subject. This post-condition formalizes our intuition of how Update() should behave.

• Lines 20–28 Application-level methods defined by the object playing the role of Observer are required to preserve the subject reference. Additionally, these methods must leave the state of the observer in a state that is consistent with the state of the subject. Note that this requirement permits the state of the observer to be modified, so long as the new state is consistent with the state of the subject. Thus, if the observer were a displayable object, we might iconify it — which would clearly change its state. However, this change would be

---

\(^{18}\)As we have already seen, some roles can be played by multiple player objects. Hence, there might be cases where it is necessary to identify a particular player within a larger set of players participating in the same role. In those cases, the ::player notation is supplemented with a selection clause that selects the object based on value and/or identity information. In those cases where the selection clause matches multiple player objects, a player is nondeterministically selected from the matching set.
permitted, provided that the definition of Consistent was such that the iconified state of the observer was considered to be consistent with the subject’s current state.

3.7.2 Memento Pattern

Following the format of the previous example, the design pattern contract corresponding to the Memento pattern is split into four separate listings. Listings 3.6 and 3.7 present the pattern-level portion of the contract. Listing 3.8 presents the role-level specification corresponding to the Originator role. Finally, Listing 3.9 presents the role-level specification corresponding to the Memento role. As before, the individual contract elements are discussed in the subsections following each listing.

Explanation of Listings 3.6 and 3.7

- Lines 3—5. The contract corresponding to the Memento pattern defines two roles, each of which corresponds to a participant type discussed in Chapter 2. Exactly one object must play the Originator role, while any number of objects may play the Memento role. Note that the contract does not define a role corresponding to the Caretaker participant type. As there are no responsibilities associated with this type of participant, and the pattern’s reward is not expressed over the state of such participants, the corresponding role is omitted.

- Lines 7—10. There are no state components — either at the role-level or the pattern-level — required to support an instance of the Memento pattern. The only identified state component is orS, an auxiliary variable associated with objects playing the Memento role. Again, as an auxiliary variable, orS does not represent a state requirement. Instead, this variable is used in the Memento
pattern_specification
Memento {
oles:
  Originator[1]
  Memento[*]

state:
  Originator: null
  Memento: Originator.ασ orS [auxiliary_variable]

pattern: null

auxiliary_concepts:

relation: ValidCopy(Originator.ασ, Memento.ασ)
relation: SameCopy(Memento.ασ₁, Memento.ασ₂)
relation: Restored(Originator.ασ₁, Originator.ασ₂)
relation: Reset(Originator.ασ₁, Memento.ασ, Originator.ασ₂)

constraint:
  (SameCopy(Memento.ασ, Memento.ασ))
  ∧
  (SameCopy(Memento.ασ₁, Memento.ασ₂)
   ⇒ SameCopy(Memento.ασ₂, Memento.ασ₁))
  ∧
  ((SameCopy(Memento.ασ₁, Memento.ασ₂)
    ∧ SameCopy(Memento.ασ₂, Memento.ασ₃))
   ⇒ SameCopy(Memento.ασ₁, Memento.ασ₃))
  ∧
  ((ValidCopy(Originator.ασ, Memento.ασ₁)
    ∧ SameCopy(Memento.ασ₁, Memento.ασ₂))
   ⇒ ValidCopy(Originator.ασ, Memento.ασ₂))
  ∧
  ((ValidCopy(Originator.ασ₁, Memento.ασ)
    ∧ Reset(Originator.ασ₂, Memento.ασ, ασ₃))
   ⇒ Restored(Originator.ασ₁, Originator.ασ₃))

pattern_instantiation:

requires:
  enrolling_as Originator:
    requires:
      true
    defines:
      ValidCopy(Originator.ασ, Memento.ασ)
      Reset(Originator.ασ₁, Memento.ασ, Originator.ασ₂)
      Restored(Originator.ασ₁, Originator.ασ₂)
  defines:
    null

Listing 3.6: The Memento Pattern: Pattern-Level Specification (1 of 2)
enrolling_as Originator:
    requires:
        false
    defines:
        null

disenrolling_as Originator:
    requires:
        false

enrolling_as Memento:
    requires:
        (player \notin \textit{enrolled\_as}(Originator))
        \land (player \notin \textit{enrolled\_as}(Memento))
        \land (\exists k : ((\textit{cs}\text{Originator}\::\textit{player}[k].method = \text{CreateMemento})
        \land (\textit{cs}\text{Originator}\::\textit{player}[k].return = player)
        \land \text{SameCopy}(\textit{cs}\text{Originator}\::\textit{player}[k].return.\alpha \sigma, \\
        player.\alpha \sigma)))
    defines:
        \text{SameCopy}(Memento.\alpha \sigma_1, Memento.\alpha \sigma_2)

disenrolling_as Memento:
    requires:
        true

invariant:
    \forall mem \in \textit{enrolled\_as}(Memento) :
        : \text{ValidCopy}(mem.orS, mem.\alpha \sigma)
}

Listing 3.7: The Memento Pattern: Pattern-Level Specification (2 of 2)
role-level specification to record the value of the originator’s application-level state at the appropriate point in the system’s execution. Specifically, at the point a memento is created by the originator, the orS variable associated with that memento records the value of the originator’s application-level state. The value of this variable — according to the role-level specification presented in Listing 3.8 — is preserved throughout the memento’s lifetime. As we will soon discuss, the orS variable plays a central role in expressing the reward for applying the pattern correctly.

• Lines 12–33 In the original pattern characterization, the authors’ note that a memento is not required to store the full state of the originator. Indeed, the type of state stored by each memento need not match the originator’s state. The important requirement is that each memento must store the appropriate information required to restore the state of the originator to the value that it possessed at the point the memento was created.

What, then, does it mean for a memento to be a valid copy of an originator? Since the answer varies from one instance of the pattern to another, ValidCopy is listed as an auxiliary concept. Given the application-level state of the originator, and the application-level state of a memento, ValidCopy indicates whether the memento’s state is faithful to the originator’s state. That is, the concept indicates whether the memento could be used to restore the state of the originator to the specified originator value. At the point this concept is used in Listing 3.8, the definition supplied by the designer is used to determine whether the memento
created by the originator contains a valid copy of the originator’s application-level state. As we will soon discuss, this concept is also used in expressing the pattern invariant.

Next, recall that the pattern description suggests that once the state of a memento has been set by the originator, its state cannot be modified. However, there is no need for such a rigid restriction. If, for example, an object playing the role of Memento provides state in support of additional areas of concern, changes to these additional state components are not material to the correctness of the pattern’s implementation. Moreover, even if changes are made to material components, not all changes should be disallowed. In particular, those state modifications which leave the memento in a state that preserves the same essential originator information as before the change should not be precluded.

Of course, what it means for one memento value to contain the same essential information as another memento value varies from one instance of the pattern to another. Indeed, the notion may vary per memento within a single pattern instance. Therefore, SameCopy is listed as an auxiliary concept, definitions of which are supplied at the point of memento enrollment. Given two values for the application-level state of a memento, SameCopy indicates whether each memento value contains the same essential originator information. At the point this concept is used in Listing 3.9, the definition supplied by the designer is used to determine whether the memento methods preserve the essential information stored by the memento.

Now suppose that the state of the originator at some point in time is o1, and that at this point, the originator creates a memento with state m1. At some
later point in time, when the state of the **originator** is \( o_2 \), the memento might be used to restore the state of the **originator**. According to the original pattern description, \( o_1 \) and \( o_2 \) should be identical. However, this requirement is too rigid. Consider, for example, if the **originator** represents an image rendered by a graphics application. At particular points in time, the rendered image might be captured as a lossy *jpeg* compression of the original. When the memento which encapsulates this *jpeg* is used to restore the **originator**, the restoration will not have 100% fidelity — but it might be good enough.

Since the precise meaning of *restoration* is specific to the pattern instance in question, **Restored** is listed as an auxiliary concept. Given two values for the application-level state of the **originator**, **Restored** indicates whether the second value is a proper restoration of the first. As we will soon discuss, this concept is used in determining whether the **SetMemento()** method properly restores the state of the **originator**.

Continuing the example, the code fragments presented in the original description — and reproduced in Chapter 2 — indicate that only \( m_1 \) is used in restoring the state of the **originator**. In the informal commentary, however, the authors note that the state of the **originator** at the point of restoration may also be material to the restoration process. That is, in restoring the state of the **originator** to \( o_1 \), the **SetMemento()** method may have to “merge” the information contained in \( o_2 \) with the information contained in \( m_1 \). The **Reset** auxiliary concept captures the relation that must hold over \( o_1, m_1, \) and \( o_2 \). At the point the concept is used in Listing 3.8 the definition supplied by the designer is used to determine whether
the `SetMemento()` method resets the state of the `originator` to an appropriate value.

Finally, the contract specifies the constraints imposed on definitions for these concepts. The first three clauses of the outermost conjunct require the `SameCopy` relation to be an equivalence relation. Reflexivity is required since if a `memento` is not modified, it should certainly contain the same essential information about the `originator`. Similarly, if `memento m1` contains the same essential information as `memento m2`, `m2` must contain the same essential information as `m1`. To understand the motivation for requiring transitivity, note that every `memento` method is required to preserve — according to the definition of `SameCopy` — the application-level state of the `memento`. Requiring transitivity guarantees that every `memento` stores, throughout its lifetime, the same essential information as when it was first created. If transitivity were not required, individual method calls could erode the information stored by a `memento`.

The fourth clause of the outermost conjunct binds the relationship between the `ValidCopy` and `SameCopy` relations. Concretely, the conjunct requires that if a particular `memento` state — say `m1` — is a valid copy of some `originator` state — say `o1`, then all other `memento` states that contain the same essential information as `m1` must also be valid copies of `o1`. This constraint guarantees, given the `Memento` role-level specification in Listing 3.9, that every `memento` will remain a valid copy of the `originator`'s state at the point the `memento` was created.

The final clause really captures the essence of the pattern. The clause states that if a `memento` is a valid copy of some `originator` state `os1`, and is later
used to reset the originator from os2 to os3, the states os1 and os3 must satisfy the Restored relation. Along with the other three constraints, and given the Originator role-level specification in Listing 3.8, the final constraint guarantees that the mementos created at each point can later be used to restore the state of the originator as appropriate to the pattern instance.

- **Lines 35—45** As mentioned previously, the contract requires exactly one object to play the role of Originator. Therefore, at the point of pattern instantiation, the designer is required to supply the object that will play this role. Along with this player object, the designer must supply appropriate definitions for ValidCopy, Reset, and Restored.

- **Lines 47—51** To prevent additional objects from enrolling in the Originator role once an instance of the pattern has been created, the contract specifies the impossible enrollment condition of false.

- **Lines 53—55** Like the enrollment condition, the contract specifies false as the condition that must be satisfied for an originator to disenroll. Therefore, if the originator object disenrolls, the contract is violated, and the pattern instance in question is implicitly destroyed.

- **Lines 57—66** For a player to enroll as a memento, the contract requires that the object not already be enrolled in the current pattern instance. Moreover, the object must have been created by the originator’s CreateMemento() method. And finally, the application-level state of the to-be memento must contain the

---

19Note that cs_{Originator::player} refers to the object call sequence associated with the player enrolled in the Originator role.
same essential originator information as when it was first created. This final clause prevents a memento object from disenrolling\(^\text{20}\), modifying its state to a value that does not represent any valid copy of the originator’s state, and then re-enrolling. Without such a condition, the pattern invariant could not be guaranteed.

- **Lines 68—70** Memento objects are free to disenroll at any point. The condition for disenrolling from the Memento role is therefore trivially specified as true.

- **Lines 72—74** The intent of the Memento pattern is to externalize the state of the originator, so that at some later point, the externalized state can be used to restore the originator to its previous value. The reward for applying the Memento pattern is therefore a guarantee that every memento created by the originator is a valid copy of the originator’s state at the point the memento was created. Combined with the constraint imposed on ValidCopy, Reset, and Restored, Listing 3.8 specifies that at any point, one of these mementos can be used to restore the originator to the appropriate value.

Explanation of Listing 3.8

- **Lines 3—4** There are no initial conditions imposed on objects enrolling in the Originator role. Hence, the initial condition is specified as true.

\(^{20}\)Recall that player objects are absolved of their role-level responsibilities once they disenroll from a pattern instance. Therefore, if a memento object were to disenroll, its methods would not be required to preserve — according to the definition of SameCopy — the application-level state of the memento.
role_specification
Originator {
  initially:
    true

  invariant:
    true

  methods:
    Memento CreateMemento():
      requires [can_strengthen]:
        true
      preserves:
        ασ
      ensures:
        (new result)
          ∧ ValidCopy(ασ, result.ασ)
          ∧ (result.or$ := ασ)
      ensures_call_sequence:
        true

    void SetMemento(Memento mem):
      requires [can_strengthen]:
        true
      preserves:
        mem.ασ
      ensures:
        Reset(ασ@pre, mem.ασ, ασ)
      ensures_call_sequence:
        true

  others:
    requires [can_strengthen]:
      true
    preserves
      null
    ensures
      true
    ensures_call_sequence:
      true
}

Listing 3.8: The Memento Pattern: Role-Level Originator Specification
• **Lines 6—7.** Like the initial condition, the role-level invariant is trivially specified as true.

• **Lines 10—20.** Recall that the `CreateMemento()` method is used to create a new memento that records the current state of the originator. The post-condition specifies that the memento returned by the method must be a new object, and that it must contain a valid copy of the originator’s state at the point the method was called. Moreover, the post-condition includes an assignment clause that stores the application-level state of the originator in the memento’s orS auxiliary variable. Again, this clause represents a “thought action”, and does not represent a requirement that must be satisfied by the designer. As we have already seen, this variable allows us to refer to the value of the originator’s application-level state at the point the memento was created, and plays a fundamental role in expressing the pattern invariant.

Note that in light of the enrollment conditions specified in Listing 3.7, the return type of this method presents an interesting question. For an object to be returned by this method, it must be enrolled in the Memento role. However, the enrollment condition requires that for a player object to enroll as a Memento, it must be the object returned by the `CreateMemento()` method. So then, within the body of `CreateMemento()`, how can an object enroll to play this role?

The problem is not peculiar to the Memento pattern, and will arise in any pattern that requires a participant object to be created by a particular method. The problem is manifested because of an implicit assumption about how call sequence elements are recorded. In particular, we have implicitly assumed that
the call sequence entry corresponding to a particular method call is automatically added to the call sequence at the point control is returned to the caller. However, this assumption prevents us from imposing the types of constraints required to specify the Memento pattern.

To overcome this problem, we introduce a slight wrinkle in our assumptions about the operation of the call sequence. We assume that the call sequence entry corresponding to a method call is added to the call sequence just before control is returned to the caller. At these points, where no further statements will be executed before returning, patterns may be instantiated, new players may enroll, and existing players may disenroll. The constraints imposed at these points are able to specify properties involving the method call that is about to return.

- **Lines 22–30** The post-condition of SetMemento() specifies that the method must reset the state of the originator according to the definition of Reset supplied by the designer. If these implementation responsibilities are satisfied, the designer can conclude that SetMemento() restores — according to the definition of Restored — the state of the originator as appropriate to the value it possessed when mem was created. This conclusion is based on the pattern invariant, and the concept constraint imposed on ValidCopy, Reset, and Restored.

- **Lines 32–40** The pattern contract does not impose any requirements on the application-level methods of the object playing the role of Originator.

---

21 This of course excludes a possible return statement.
role_specification
Memento {
    initially:
        true
    invariant:
        true
    others:
        requires [can_strengthen]:
            true
        preserves
            orS
        ensures
            SameCopy(α@pre, ασ)
        ensures_call_sequence:
            true
    }

Listing 3.9: The Memento Pattern: Role-Level Memento Specification

Explanation of Listing 3.9

- **Lines 3–4**  As in the Originator role specification, there are no initial conditions imposed on enrolling objects. The initial condition is therefore specified as true.

- **Lines 6–7**  Like the initial condition, the role-level invariant is trivially specified as true.

- **Lines 9–17**  Since there are no role-level methods defined, the conditions imposed by this specification apply to every player method. The requirement is that at the termination of each method, the post-conditional value of the memento must contain the same essential information as the pre-conditional value. More concretely, the specification requires that the pre-conditional and
post-conditional memento values satisfy the SameCopy relation. Combined with the constraints imposed on SameCopy and ValidCopy, this guarantees that each memento will retain, throughout its lifetime, the same essential originator information as when it was first created.

**GetState() and SetState() Method Omissions.** Recall that the UML diagram presented as part of the original pattern description identifies a GetState() method for reading the state of the memento, and a SetState() method for writing the state of the memento. According to the original pattern description, the SetState() method is used by CreateMemento() to store information about the originator within the memento. Similarly, the GetState() method is used by SetMemento() to retrieve information from the memento that is used to restore the originator’s state. These methods are not specified as part of the Memento role-level specification, as there doesn’t seem to be a motivation for explicitly specifying their behavior.

First, consider the GetState() method. The important condition is that the implementation of SetMemento() satisfies its specification. As long as the interface exposed by the memento allows SetMemento() to retrieve the appropriate information without violating the application-level method constraints imposed by the contract, there is no need to explicitly specify the GetState() method. Indeed, doing so would compromise the pattern’s flexibility, since every memento would then be required to provide a GetState() method — even when other “getter” methods would be more appropriate. In short, the GetState() method is not a part of the pattern’s recurring structure.

The motivation for omitting the SetState() method is not to avoid a compromise in flexibility, but rather, to simplify the Memento role-level specification. According
to the original characterization, the `SetState()` method is only used by the originator’s `CreateMemento()` method in storing its state when a memento is being created. An alternative approach is to supply the appropriate state information to the memento’s constructor. Of course, the omission of the `SetState()` method prevents memento objects from being reused, since application-level methods are prevented from arbitrarily setting the state of the memento. This is not an issue, however, since the pattern contract requires the `CreateMemento()` method to return a new memento each time it is called.

### 3.8 Formalization Benefits

Like any complex specification, developing a design pattern contract is a challenging task. Fortunately, the task is only performed once per pattern. When the contract is complete, system designers benefit from the formalization that it provides whenever an instance of the corresponding pattern is applied.

One of the two principal advantages of the formalization approach is the precision that it brings to the pattern’s description. A contract precisely specifies the requirements that must be satisfied by the system designer, as well as the system properties that are guaranteed by satisfying those requirements. Second, and more surprising, a design pattern contract often provides more flexibility than the original pattern description. This is a particularly satisfying result given that precision and flexibility are generally considered to be incompatible, competing concerns.

#### 3.8.1 Improved Precision

The precision advantage offered by our approach is especially evident in the case of the Observer pattern. Examining the original pattern characterization, it is clear that
the designer must settle on a definition of what it means for a subject to be modified. Similarly, it is clear that the designer must settle on a definition of what it means for an observer to be consistent with the subject. However, the potential incompatibility between these definitions is not discussed. New designers — the primary audience for design pattern catalogs — could easily use definitions that allow for the subject to become inconsistent with one or more observers. As a consequence, the whole point of applying the pattern could be undermined. The compatibility requirement is made precise in the Observer pattern contract, reducing the possibility that this error will be made.

Next, consider the case of an attaching observer. Recall that Attach() is required to invoke the observer’s Update() method to ensure that the observer’s state is consistent with that of the subject. Again, if the designer neglects to include such a call in the body of Attach(), the pattern’s purpose is compromised. The original pattern description makes no mention of this requirement. The responsibility is clear, however, in our pattern contract.

Finally, note that the original pattern description does not include a discussion of the changes that can be made to an observer independent of the subject. The informal characterization therefore fails to capture the important requirement that application-level observer methods must maintain consistency between the subject and the observer. While this responsibility might be obvious to some designers, capturing it explicitly in the contract makes it obvious to all designers.

22Note that we use the term “reduce” rather than “eliminate” since there is never a guarantee that the designer will correctly satisfy the contract requirements — even when they’re stated precisely.

23That is, all the designers familiar with our contract formalism.
Thus, even for a relatively simple pattern such as Observer, developing the corresponding pattern contract was helpful in identifying and resolving ambiguities in the informal description. Indeed, it was only as we were developing the contract that many of these ambiguities became evident. Future designers are now able to benefit from the precise characterization provided by our contract.

### 3.8.2 Improved Flexibility

The flexibility advantage provided by our approach is particularly clear in the Memento example. Consider the pattern’s canonical application in a graph editing system [Gamma et al. 1995]. In this scenario, the state of the **originator** consists of the various nodes and edges of a graph being modified by the user. After each operation on the graph, the **originator** generates a **memento** that can later be used to restore the graph to a previous state. In the original description, the notion of restoration requires each node and each edge to be at exactly the same positions they occupied originally. The Memento contract certainly allows for this; the designer need only supply the appropriate definition for **Restored**.

On the other hand, another notion of restoration might only require that the nodes be moved to their original locations, so long as the connectivity of the graph is the same as it was previously. An even weaker notion might require the nodes to be restored to their original locations without imposing any conditions on the edges. All of these possibilities are accommodated by the contract. Again, the designer is simply required to supply the appropriate concept definition.
The **Restored** concept introduces a point of variation that does not seem to be present in the original pattern description. In a sense, the introduction of this concept *encourages* the discovery of new notions of restoration, increasing the pattern’s flexibility. Of course, once these additional dimensions of flexibility are discovered, it might be desirable to update the pattern’s informal description. The fact would remain, however, that it was the pattern contract that led to the identification of these new dimensions.

It is worth mentioning, however, that some might object to the claim that **Restored** represents a new point of variation. Instead, they might argue, the concept identifies a point of variation that is *implicitly* allowed by the pattern’s description. After all, the original description describes canonical cases; designers must generalize over these examples.

This is certainly a possibility. It may well be that the authors intended for designers to use various definitions of restoration. However, making these points of variation *explicit* in the contract encourages designers to vary the pattern at these points. In a sense, each concept serves a guide to the designer, indicating a dimension of variability. Without these explicit cues, the novice designer is unlikely to determine the appropriate ways in which particular patterns can be specialized.

### 3.9 Chapter Summary

A *design pattern contract* is a specification instrument used to precisely characterize a design pattern. The contract consists of a *responsibilities* component and a *rewards* component. The responsibilities component identifies those properties that must be satisfied by the designer to correctly apply an instance of the pattern. The
rewards component specifies the properties that the resulting system is guaranteed to exhibit if the contract responsibilities are indeed met.

Since a design pattern describes a behavioral and structural slice through a system, the corresponding contract must characterize only those aspects of the participating objects that are relevant to the pattern’s implementation. To allow for this type of partial specification, each contract models the participating objects through a set of *roles*. Each role represents a projection of an object with respect to those aspects that are material to its responsibilities in the pattern. The contract characterizes the state and behavior required to support each role without over-specifying the objects that ultimately enroll to play them.

Every pattern contract is parameterized by a set of *auxiliary concepts* that abstract over the points of variation among instances of the same pattern. These concepts are used in specifying the pattern’s behavior without compromising the pattern’s flexibility. Indeed, auxiliary concepts encourage flexibility by making the pattern’s dimensions of variability explicit. At appropriate points in the pattern’s lifetime, the designer is required to supply definitions for these concepts, allowing the pattern contract to be tailored as appropriate to the pattern instance.

The material presented in this chapter builds on earlier work described in [Soundarajan and Hallstrom 2004].
CHAPTER 4

CONTRACT MONITORING USING DYNAMICALLY GENERATED WRAPPERS

“An error the breadth of a single hair can lead one a thousand miles astray.”
— Chinese Proverb.

4.1 Chapter Overview

In Chapter 3, we introduced design pattern contracts as a formalism for specifying design patterns. In this chapter, we present an approach to monitoring whether the contracts used in developing a particular system are respected at runtime. After presenting the basic approach, we consider monitoring an instance of the Observer pattern to check whether the instance satisfies the pattern contract presented in the previous chapter. We conclude with a brief discussion of the advantages and limitations of the monitoring approach. These advantages and limitations will serve as points of comparison when we consider the alternative monitoring technique presented in the next chapter.
4.2 Detecting Contract Violations

When a system designer chooses to apply a particular pattern, the corresponding contract serves as a guide. As long as the designer satisfies the specified responsibilities, she can conclude that the pattern has been applied correctly. Moreover, she can reason about the resulting system based on the properties guaranteed by the pattern invariant. For these conclusions to be justified, however, the designer must be sure that the appropriate responsibilities have in fact been met. After all, software design is a complex task. Even when guided by a specification, system designers can — and do — make mistakes. Missing even a seemingly minor requirement can compromise the pattern invariant. If the correctness of other system objects depends on this invariant, the local failure could spread, resulting in a global system catastrophe.

One approach to addressing this concern is to require the designer to statically verify each pattern’s implementation with respect to the corresponding contract. In short, the designer would be required to show that the verification conditions outlined in the previous chapter were indeed satisfied. Depending on the reliability requirements of the system in question, the verification process could be formal, semi-formal, or informal.

Unfortunately, many practitioners are reluctant to introduce a verification step into their development process. The reluctance appears to stem from two principal arguments. First, even when done informally, the verification process can be time-consuming. Although the reduction in time spent on maintenance activities is likely to outweigh the time spent on verification, it does delay the initial deployment of the system. In many cases, the economic importance of time-to-market makes this delay infeasible. Second, software verification requires a background in mathematics that
many software developers do not possess. The training time required to develop the necessary skill set might not be available. As one of the principal goals underlying our work is accessibility to practitioners, requiring a verification step is therefore unacceptable.

An alternative measure, which might be used in combination with verification procedures, is to monitor whether the contracts used in developing a particular system are respected at runtime. By instrumenting the system with the appropriate runtime checks, contract violations can be reported as part of the standard testing and debugging process. Of course, no amount of testing can guarantee that the patterns in question have been applied correctly. However, if appropriate test cases are selected, testing can reveal the presence of the most common errors. More important, software testing is widely accepted as a standard part of programming practice — software practitioners are more likely to apply such techniques.

The code used to monitor a system should be deployable on a per contract basis. We refer to these units of deployment as “contract monitors”. Like the contracts being checked, each monitor must be general enough to accommodate the variations that occur across instances of the corresponding pattern. At the same time, each monitor must be customizable. It must be possible to tailor the assertions that will be checked per pattern instance. Given a library of such monitors, the designer should be able to quickly instrument a system as appropriate to the patterns used in constructing it.
4.3 The Monitoring Approach

We now turn our attention to the design of a monitoring library that satisfies the aforementioned desiderata. Before proceeding, however, it is useful to recall that when verifying a pattern’s implementation, the verification conditions materialize at three points: (i) at the point the pattern is instantiated, (ii) at the point an object enrolls in an instance of the pattern, and (iii) at the point an object disenrolls from an instance of the pattern. Recall, however, that these points are not explicit in the program code — they exist only in the mind of the designer. To make the designer’s verification responsibilities clear — as discussed in Chapter 3 — these points must be made explicit by introducing suitable pattern code. In our monitoring approach, this pattern code takes the form of standard method calls to a contract monitoring class library.

4.3.1 Contract Monitors

The contract monitoring library consists of a number of “monitor factories” — each realized as a single class that corresponds to a particular pattern type. A monitor factory is responsible for creating and maintaining the contract monitors responsible for monitoring the specialized contracts associated with individual pattern instances. When a pattern is conceptually instantiated, the designer is required to place an instantiation call to the appropriate factory class. This call results in the creation of a new contract monitor that monitors the contract associated with the pattern instance being created. As part of the instantiation call, the designer is required to supply an identifier that uniquely identifies the pattern instance — and therefore,

\[\text{Note that class methods are referred to as “static methods” in languages like C++, Java, and C#}.\]
uniquely identifies the contract monitor. Later, when an object enrolls to play a role in a pattern instance, the designer is required to place an enrollment call to the corresponding contract monitor. The monitor is accessed through the factory that created it using the appropriate pattern identifier. A similar call is required when an object disenrolls from the pattern instance. The basic monitoring class structure is illustrated in Figure 4.1.

Recall that at the point of pattern instantiation, as well as at the point of enrollment, the designer is required to supply a number of definitions that tailor the pattern contract to the pattern instance in question. These definitions are material to the verification conditions that must be dispatched, and are therefore material to the assertions that must be checked by the monitoring code. To accommodate this variation, each monitor mirrors the structure of the contract that it checks. More

\[\text{As we have already seen, the point of enrollment may coincide with the point of pattern instantiation. In such cases, the instantiation call additionally serves as an enrollment call. Hereafter, when we refer to an enrollment call, we are implicitly considering the cases where the instantiation call serves a dual purpose.}\]
concretely, each monitor is parameterized by the definitions required to tailor the monitoring behavior to a particular pattern instance. Therefore, when placing an instantiation call, the designer is required to supply executable definitions for the appropriate pattern-level abstraction functions, as well as any pattern-level auxiliary concepts. Similarly, at the point of enrollment, the designer is required to supply executable definitions for the appropriate role-level abstraction functions, as well as any role-level auxiliary concepts. The objects supplied by the designer are used by the contract monitor to check that the contract responsibilities are satisfied at the relevant points.

Note that the designer is not required to explicitly supply the object types corresponding to the roles identified in the contract, nor to explicitly provide the application-level state model corresponding to each player. As we will see when we consider the example presented in Section 4.4, there is no need to explicitly supply this information, as it is implicitly provided by the auxiliary concept and abstraction function definitions. Explicit definitions are, however, required for the mappings which specify how each player-level method corresponds to its role-level analogue. We will soon discuss these definitions in more detail.

### 4.3.2 Role Monitors

We have already stated that each monitor checks that the contract responsibilities are satisfied at the appropriate points. But what exactly does “at the appropriate points” mean? When a pattern is instantiated, the call to the monitor factory can directly check that the necessary number of players have been provided, and that
definitions for the appropriate auxiliary concepts have been supplied\textsuperscript{26}. Similarly, when an object enrolls in a pattern instance, the contract monitor can directly check that the appropriate state conditions are met. More precisely, the monitor can use the executable definitions supplied by the designer to determine whether the player object satisfies its role-level invariant, role-level initial value constraint, and any additional constraints imposed by the contract’s enrollment clause. The case is analogous for a disenrolling object. Using the executable definitions supplied by the designer, the monitor can directly check that any “disenrollment” conditions have been met.

But what about the player object’s behavioral responsibilities? Clearly the contract monitor cannot determine at the point of enrollment whether a particular player method satisfies its role-level responsibilities. Instead, the player object must itself be monitored to ensure that each invocation on the player is consistent with the behavior required by its role-level specification. To achieve this monitoring, the approach that we take is based on the notion of a “checking wrapper” [Edwards et al. 1998]. The essential idea behind this approach is to encapsulate each player object within a wrapper that exposes the same interface as the player being wrapped. This wrapper can then be used in place of the original object without having to modify the client code. When a call is placed on the object, the wrapper intercepts the call, and checks that the appropriate pre-conditions and invariants are satisfied. The wrapper then delegates the call to the underlying player object, and again intercepts the control flow before returning to the client. At this point, the wrapper checks that the appropriate post-conditions and invariants are satisfied. Any violations detected by the

\textsuperscript{26}Note that it is not possible for the monitor factory to determine whether the concept definitions supplied by the designer satisfy any constraints imposed on them by the contract. We must therefore assume that the definitions supplied by the designer are always consistent with the contract to be checked.
wrapper will be reported, alerting the programmer that the pattern contract has been breached. Hereafter, we refer to this type of checking wrapper as a “role monitor”. The basic structure of a role monitor is illustrated in Figure 4.2.

Recall that the lifetime of a role is not necessarily identical to the lifetime of the object that plays it. A player object is bound by its role-level specification only while it is enrolled in an instance of the corresponding pattern. Hence, a player object need not be monitored throughout its lifetime. It is only while the player object is enrolled in an instance of the pattern that it should be wrapped by a role monitor. Therefore, at the point of enrollment, the designer is required to exchange the enrolling object for the role monitor that wraps the object. More specifically, the designer is required to pass the enrolling object to the enrollment method of the appropriate contract monitor. The contract monitor itself behaves as a factory, using the supplied executable definitions to construct a role monitor that wraps the enrolling player. The wrapped player object — which appears to client objects as the player itself — is returned by the enrollment method, and is then swapped with the
original player object. Since the interface provided by the role monitor is identical to the interface provided by the enrolling object, the exchange is transparent to the client code. When the player disenrolls, the process is reversed. The corresponding role monitor must be passed to the “disenrollment” method of the contract monitor that created it. The method returns the object wrapped by the role monitor, and this object is then swapped back into service.

An important point worth noting is that once an object enrolls in a pattern instance, any aliased references to the player object must also be exchanged for the reference to the role monitor that wraps it. Otherwise, client objects could bypass the role monitor when accessing the player, and any resulting contract violations might not be detected. Similarly, when the player object disenrolls, these references should be exchanged for the original player reference.

Another point worth noting is that a role monitor may have to introduce additional state to support its monitoring task, or perhaps to support the monitoring task of another monitor associated with the same pattern instance. In general, this will be necessary when one or more role monitors are required to check that call sequence conditions are being respected. Consider, for example, the Observer pattern, and the requirement that the subject’s Notify() method invoke Update() on each attached observer. One approach to checking this requirement is to record the sequence of incoming calls from within each observer’s role monitor. The role monitor associated with the subject could clear these entries before Notify() is called, and upon the method’s termination, check that the call sequence contains the appropriate invocations. Of course, an alternative would be to introduce state components corresponding to the
auxiliary variables necessary to capture this same requirement. We will examine this alternative in more detail when we consider the example in Section 4.4.

Before turning our attention to that example, however, there is one last wrinkle that needs to be addressed. Consider once again the structure of a role monitor, as illustrated in Figure 4.2. The interface exposed by the monitor must precisely match the interface exposed by the player that it encapsulates. If this were not the case, the role monitor could not be used interchangeably with the original player. However, player objects are likely to expose interfaces that vary from one object to the next. The variation stems from two causes. First, some player objects will provide application-level methods that other player objects do not provide. This is a simple consequence of the fact that objects participate in different areas of concern. Second, a role-level specification does not precisely specify the syntactic interface that must be provided by an enrolling player. Some player objects will use the method signatures suggested by the role-level specification, while others will satisfy the contract with customized signatures. Indeed, this is why the designer is required to supply the appropriate method and parameter mappings when verifying that a particular player object satisfies its role-level specification.

To accommodate the syntactic variation among player interfaces, the type of each role monitor is generated dynamically at the point of enrollment. The enrollment method first reflects upon the enrolling object to determine the interface that the new role monitor must provide. This meta-data is then used to dynamically construct a monitor type that exposes an identical interface. This dynamically generated type is used to create a role monitor tailored to the enrolling player.
Of course, the purpose of the new role monitor is to check whether the player object satisfies its role-level method responsibilities. Hence, there must be a way to determine how each player method corresponds to its role-level counterpart. If this were not the case, how would the monitor determine the conditions that should be checked for a particular method call? To address this issue, the designer is required to supply meta-data at the point of enrollment that characterizes how each player method maps to its role-level analogue. Moreover, the designer must supply meta-data characterizing the correspondence between the method parameters. This meta-data is used by the role monitor to map the relevant assertion checks to the appropriate player methods and parameters. The structure of a role monitor with mapped assertion checks is illustrated in Figure 4.3.

4.3.3 The Dynamic Proxy API

When creating a new role monitor, the enrollment method is not directly responsible for generating any source code. Instead, the creation of the role monitor is
left to the Dynamic Proxy API — a standard element of the Java class library, as well as Microsoft’s .NET Framework\textsuperscript{27}. The API enables the creation of a proxy object [Gamma et al. 1995] based on a list of interfaces to be supplied by that proxy. The proxy’s type is generated dynamically, and provides “stub” implementations for each of the methods that it exposes. Each stub, when invoked, captures the method’s signature, as well as any arguments that were passed during the call. This information is marshaled into an “invocation request”, which is forwarded by the proxy to an “invocation handler”. As its name suggests, an invocation handler is an object that handles invocations on behalf of the proxy. The invocation handler must be supplied by the designer when the proxy is created.

Every invocation handler exposes an “invocation method” that accepts an invocation request as argument. When an invocation handler receives a request — through a call to the invocation method — it un-marshal the request, performs the appropriate processing, and returns any results back to the associated proxy.\textsuperscript{28} The proxy in turn, returns the results to the client, who is unaware of the additional levels of indirection. From the client’s perspective, the proxy appears as a standard implementation of the interfaces that it provides. The structure of a dynamically generated proxy is illustrated in Figure 4.4.

The Dynamic Proxy API is particularly well-suited to handling the creation of role monitors. In our approach, the enrollment method reflects on the enrolling player to determine the interfaces that it implements. This interface meta-data is used by

\textsuperscript{27}In the .NET Framework, the API is not referred to as the “Dynamic Proxy API”. Instead, the equivalent functionality is provided as part of the .NET “Remoting” namespace.

\textsuperscript{28}Note that an invocation handler is not required to un-marshal the invocation request. In some cases, for example, the proxy might simply log the method calls through the proxy, and return null as the return value.
the Dynamic Proxy API to construct a new proxy that exposes the same interfaces as the enrolling player. The invocation handler used by the proxy implements the checking behavior appropriate to the role in which the player is enrolling. More specifically, when the invocation handler receives an invocation request, it performs the appropriate assertion checks before and after delegating the incoming call to the original player object. The meta-data that describes how each player method corresponds to its role-level counterpart is used by the invocation handler to determine which assertion checks to perform when a marshaled request is received. In effect, the proxy and the invocation handler together serve as a dynamically generated checking wrapper that provides mapped assertion checking. The use of the Dynamic Proxy API in creating a new role monitor is illustrated in Figure 4.5.

4.4 Example: The Observer Pattern

To illustrate the monitoring approach, we now turn our attention to an example. Section 4.4.1 presents the monitor corresponding to the Observer pattern contract.
Figure 4.5: Using the Dynamic Proxy API
presented in Chapter 3. Section 4.4.2 considers an application of this monitor to a
typical Observer pattern instance.

4.4.1 Observer Pattern Contract Monitor

We begin by considering the executable definitions that must be supplied by the
designer to tailor the contract monitor to a particular pattern instance. In the case
of the Observer pattern, there are no pattern-level state components or auxiliary
concepts, so the only executable definitions that must be provided are those associated
with the pattern roles. The $\text{SubjectDefs}$ interface, shown in Listing 4.1, defines the
methods that must be supplied when an object enrolls in the $\text{Subject}$ role. These
methods correspond to the abstraction functions, concept definitions, and mappings
that must be defined when verifying the pattern’s implementation. Note that these
methods need not be provided by the enrolling object. Instead, the designer will
supply an independent implementation of this interface at the point of enrollment.
Similarly, the $\text{ObserverDefs}$ interface, shown in Listing 4.2, defines the methods that
must be supplied when an object enrolls in the $\text{Observer}$ role. The interface methods
are described in the subsections that follow.

Explanation of Listing 4.1

- **Line 3**. The $\text{getSC.Observers()}$ method corresponds to the abstraction function
  that maps the player’s state to the role-level $\text{observers}$ component. Given the
  player object enrolled in the $\text{Subject}$ role, this method must return the set of
  attached $\text{observers}$.

$^{29}$Note that the methods corresponding to abstraction functions — such as $\text{getSC.Observers()}$ —
are used by the checking invocation handler to record the values of individual state components before
and after invoking methods on the underlying player object. These recorded values are used by the
handler to check that the appropriate post-conditions are satisfied. Since the post-conditions are
public interface SubjectDefs
{
    public Set getSC_Observers(Object player);
    public Object getSC_AS(Object player);
    public boolean ac_Modified(Object AS1, Object AS2);
    public String[] getMethodMap();
}

Listing 4.1: Observer Contract Monitor: SubjectDefs Interface

- Line 5 The getSC_AS() method corresponds to the abstraction function that maps the player’s state to its application-level state model. This method must return an application-level state encoding of the player passed as argument.

- Line 7 The ac_Modified() method corresponds to the Modified auxiliary concept. Given two application-level state values — as produced by getSC_AS() — this method must return true if the state has been modified, and false otherwise.

- Line 9 Finally, getMethodMap() is used to produce the map that specifies the player-level method corresponding to each role-level method. The array returned by getMethodMap() must contain two string entries. The first entry of the array must contain the name of the method corresponding to Attach(), and the second must contain the name of the method corresponding to Detach().

Expressed over pre-conditional and post-conditional values, the objects returned by these methods must not be aliased. Each method corresponding to an abstraction function must return a new object that encodes the appropriate value.

Note that in general, each array entry must contain more than a method name to uniquely identify a particular player method. In the presence of overloading, each array entry must additionally contain signature information.
public interface ObserverDefs {
    
    public Object getSC_Subject(Object player);
    
    public Object getSC_AS(Object player);
    
    public boolean ac_Consistent(Object subAS, Object obsAS);
    
    public String[] getMethodMap();

}

Listing 4.2: Observer Contract Monitor: ObserverDefs Interface

Notice that there is no entry corresponding to the Notify() method. We will later discuss the absence of this entry in more detail.

For the sake of presentation, we have omitted the method that yields the parameter map corresponding to each method. As Attach() and Detach() are the only role-level methods with parameters, they are the only methods affected by this omission. We assume that any player method corresponding to Attach() receives the attaching observer as its first parameter. Similarly, we assume that any player method corresponding to Detach() receives the detaching observer as its first parameter.

Explanation of Listing 4.2

• Line 3 The getSC_Subject() method corresponds to the abstraction function that characterizes how the _subject state component is realized. Given the player enrolled as Observer, this method must return the subject reference maintained by that player.
• **Line 5** The purpose of the `getSC_AS()` method is identical to that of the `getSC_AS()` method defined in the `SubjectDefs` interface. This method must return an encoding of the observer’s application-level state.

• **Line 7** The `ac_Consistent()` method corresponds to the `Consistent` auxiliary concept. Given the application-level state values of the subject and the observer — as produced by `SubjectDefs.getSC_AS()` and `ObserverDefs.getSC_AS()` — this method must indicate whether the subject and the observer are consistent.

• **Line 9** The purpose of `getMethodMap()` is again identical to that of the `getMethodMap()` defined in the `SubjectDefs` interface. The array returned by this method must contain only a single entry; this entry must contain the name of the player method corresponding to `Update()`. Notice that `ObserverDefs` does not define a method for producing a parameter map. Such a method is not required since none of the role-level Observer methods define parameters.

We now turn our attention to the `ObserverCM` class, which implements the contract monitor corresponding to the Observer pattern. The class additionally serves as a factory that produces `ObserverCM` instances. That is, contract monitors cannot be created directly. Instead, every monitor is created through the class methods provided by `ObserverCM`. The implementation is split across two listings. The first, Listing 4.3 presents the portion of the class that is material to its behavior as a factory. The second, Listing 4.4 presents the portion that is material to its behavior as a contract monitor.
public class ObserverCM {

    private static Map contracts = new HashMap();

    public static ObserverCM createInstance(String id, Object subject, SubjectDefs defs) {
        if (contracts.containsKey(id)) {
            System.err.println("! contract already exists.");
            System.exit(-1);
        }

        ObserverCM contract = new ObserverCM();
        contracts.put(id, contract);
        Object proxy = contract.enrollAsSubject(subject, defs);
        return proxy;
    }

    public static ObserverCM getInstance(String id) {
        /* ... contract retrieval omitted ... */
    }

}
Explanation of Listing 4.3

- **Line 3** Recall that every pattern instance — and therefore every contract monitor — is uniquely identified by a pattern identifier. The contracts data member is used to maintain the association between identifiers and contract monitors.

- **Lines 5—19** The `createInstance()` method is used to create a new contract monitor. When this method is called, the client must pass a unique pattern identifier, the player enrolling in the Subject role, and the `SubjectDefs` instance corresponding to the enrolling player. The method first checks whether the specified pattern identifier is in use. If it is, an appropriate error message is displayed and the system is terminated. Otherwise, the method creates a new monitor, and enrolls the player passed as argument in the Subject role. The enrollment method used by `createInstance()` returns a new role monitor corresponding to this player, which is then returned to the client of `createInstance()`.

- **Lines 21—22** The `getInstance()` method is used by clients to retrieve the contract monitor associated with a particular pattern instance.

Explanation of Listing 4.4

- **Lines 3—4** The default constructor is marked as private to prevent instances of the class from being constructed without calling `createInstance()`.

- **Lines 6—22** The `enrollAsSubject()` method is the enrollment method used by `createInstance()` to enroll a player in the Subject role. The method begins by

\[^{31}\] Note that since a player object can only enroll as a Subject at the point of pattern instantiation, `enrollAsSubject()` is a private method that is only called from within `createInstance()`.

104
private ObserverCM()
{

private Object enrollAsSubject(Object subject, SubjectDefs subjectDefs)
{
    if (subjectDefs.getSC_Observers(subject).size() != 0)
    {
        System.err.println("! observers set is non-empty.");
        System.exit(-1);
    }

    /* ... data member assignments omitted ... */

    Object proxy = null;
    Class[] interfaces = subject.getClass().getInterfaces();
    SubjectIH invHandler = new SubjectIH(subject, subjectDefs);
    /* ... proxy creation omitted ... */
    return (proxy);
}

public Object enrollAsObserver(Object observer, ObserverDefs observerDefs)
{ /* ... observer enrollment omitted ... */ }

/* ... disenrollment methods omitted ... */

Listing 4.4: Observer Contract Monitor: ObserverCM Class (2 of 2)
checking whether the enrolling player violates the initial state condition imposed
by the role-level specification. More specifically, the method uses the supplied
definition of getSC_Observers() to check whether the player’s _observers set is
non-empty. If the set is non-empty, the contract violation is reported and the
system is terminated. Otherwise, the method constructs a new role monitor cor-
responding to the enrolling player, and returns the monitor to createInstance().
Notice that in creating the new role monitor, the method first reflects on the
enrolling player to determine the interfaces that it supports. It then constructs
an instance of SubjectIH — the checking invocation handler corresponding to
the Subject role. The interface meta-data and the checking invocation handler
are together used to construct the proxy that serves as the player’s role monitor.

• Lines 24—26. As its name suggests, enrollAsObserver() is the enrollment
method corresponding to the Observer role. The structure of this method par-
allels that of enrollAsSubject(). First, the method checks that the appropriate
conditions are satisfied. Namely, the method checks that the player’s _subject
state component references the object enrolled in the Subject role. To retrieve
this reference, the method uses the supplied definition of getSC_Subject(). Next,
the method checks that the enrolling player is not already enrolled in the pat-
tern instance being monitored. If no violations are detected, the enrollment
method creates a new role monitor that wraps the enrolling player, and returns
the monitor to the caller.

• Line 28. The ObserverCM class additionally provides methods for player ob-
jects to disenroll from the pattern instance being monitored. These methods
essentially unwrap the player contained within the role monitor, and return the object to the caller.

We next consider the invocation handlers used by the dynamically generated proxies to provide their role monitoring functionality. Each checking invocation handler is specific to a particular role. Hence, in the case of the Observer pattern, the contract monitor uses two types of invocation handlers. The SubjectIH class implements the type of invocation handlers created by enrollAsSubject() to check the assertions appropriate to the Subject role. Similarly, the ObserverIH class implements the type of invocation handler created by enrollAsObserver() to check the assertions appropriate to the Observer role. In the next two listings, we present the key portions of the ObserverIH class. As all invocation handlers follow a common structure, the SubjectIH class shares a similar implementation.

Explanation of Listing 4.5

- **Lines 4–8** The ObserverIH class defines six data members. The first data member, observer, stores the player object being monitored. When an invocation is received by an instance of this class, the observer member ultimately services the call. The next two members, subjectDefs and observerDefs, store the definitions required to customize the handler’s checking behavior. Unlike the other data members, the last two are not provided to support instances of ObserverIH. Instead, these members are provided to support instances of SubjectIH. The updateCalled variable is set to true by ObserverIH whenever
public class ObserverIH
    implements InvocationHandler
{
private Object observer;
private SubjectDefs subjectDefs;
private ObserverDefs observerDefs;
private boolean updateCalled;
private Object lastSubjectAS;

public ObserverIH(SubjectDefs subjectDefs,
                  Object observer,
                  ObserverDefs observerDefs)
{ /* ... initialization omitted ... */ }

public Object invoke(Object proxy,
              Method method,
              Object[] args) throws Throwable
{
    String[] methodMap = observerDefs.getMethodMap();
    if(method.getName().equals(methodMap[0]))
    { return (update(method, args)); }
    else
    { return (others(method, args)); }
}
Update() is invoked on the underlying player. Similarly, at the start of every observer invocation, ObserverIH records the application-level state of the subject in the lastSubjectAS variable — using observerDefs.getSC_Subject() and subjectDefs.getSC_AS(). These two data members correspond to the auxiliary variables required to capture the call sequence requirements imposed by the Subject role-level specification.

When a call is placed to Attach() through the subject’s role monitor, the corresponding instance of SubjectIH sets the updateCalled variable of the attaching observer’s invocation handler to false. After the method has been invoked on the underlying subject, SubjectIH checks that updateCalled has been set to true. If this were not the case, Attach() must have failed to call Update() on the attaching observer. In this scenario, SubjectIH would report the contract violation and terminate the system. A similar technique is used when checking that Notify() invokes Update() on every attached observer.

Next, recall the strengthened preservation condition imposed on subject methods. Namely, the post-conditional value of the subject must be unmodified — according to the definition of Modified — with respect to its value at the point of the last call to each observer. Otherwise, temporary changes to the subject within a method could cause one or more observers to become inconsistent with the final state of the subject. The lastSubjectAS variable is used to check that this condition is satisfied. Concretely, at the end of every invocation through

\footnote{For convenience, we use the role-level method name Update() to refer to the player method that corresponds to Update().}

\footnote{Again, we use the role-level method name Attach() to refer to the player method that corresponds to Attach().}
the subject’s role monitor, the corresponding instance of SubjectIH checks that
the current state of the subject is unmodified — according to the definition of
ac_Modified() — from the state recorded in the lastSubjectAS variable contained
within each observer’s invocation handler. If the assertion check succeeds, the
designer can conclude that the preservation constraint was respected during the
call.

- Lines 15—24. The invoke() method is used by the associated proxy to signal
the handler of an invocation request. When a request is received, invoke() dispatches the call to the appropriate private checking method based on the
map produced by getMethodMap(). Each private method performs the assertion
checks appropriate to a particular role-level method before and after invoking
the actual method on the player.

Explanation of Listing 4.6

The private update() method is used by the invocation handler to service calls to
the player’s Update() method. The method begins by taking a snapshot of the pre-
conditional values required to check the player method’s post-condition. Next, the
method updates the data members used by SubjectIH in checking the call sequence
conditions discussed above. After invoking the actual method on the underlying
observer, the checking method records the appropriate post-conditional values, as well
as the value of the observer’s application-level state. The method then checks that
the three requirements imposed on the player method’s post-condition are satisfied.
First, the method checks that the _subject reference was preserved by the call. Next,

34 Again, we use the role-level method name Update() to refer to the player method that corre-
sponds to Update().
```java
private Object update(Method method, 
                        Object[] args) throws Throwable 
{
    Object sub_PRE = observerDefs.getSC_Subject(observer);
    Object subAS_PRE = subjectDefs.getSC_AS(sub_PRE);

    updateCalled = true;
    lastSubjectAS = subAS_PRE;

    Object result = method.invoke(observer, args);

    Object sub_POST = observerDefs.getSC_Subject(observer);
    Object subAS_POST = subjectDefs.getSC_AS(sub_POST);
    Object obsAS_POST = observerDefs.getSC_AS(observer);

    if((sub_PRE != sub_POST) || (!subAS_PRE.equals(subAS_POST)))
    {
        System.err.println("! subject not preserved.");
        System.exit(-1);
    }

    if(!observerDefs.ac_Consistent(subAS_POST, obsAS_POST))
    {
        System.err.println("! observer not consistent.");
        System.exit(-1);
    }

    return(result);
}

/* ... methods omitted ... */
```

Listing 4.6: Observer Contract Monitor: ObserverIH Class (2 of 2)
it checks that the application-level state of the subject was preserved. Finally, it checks that the post-conditional value of the observer is consistent — according to the definition of acConsistent() — with the post-conditional value of the subject. If any of these conditions are violated, the checking method generates the appropriate error message and terminates the system. Otherwise, the invocation completes as though it were invoked on the observer directly.

4.4.2 Sample Application

We now consider a typical instance of the Observer pattern, and present the code required to tailor the monitoring behavior to this instance. In particular, recall the pattern’s application in the context of designing user interfaces. In this context, consider a subject object that encapsulates the code required to complete some long running task. Next, consider two user interface objects that act as observers, and signal the user of the task’s progress. The first type of object renders a linear bar graph, the length of which is proportional to the task’s completion percentage. The second type of object renders a button that displays the task’s output when clicked. This button should only be enabled when the task has finished running. In this pattern instance, each observer implements the TaskListener interface. This interface exposes only two methods — one corresponding to Update(), and one used to retrieve a reference to the observer’s subject. The subject implements the Task interface, which is presented in Listing 4.7.

At the point of pattern instantiation, the designer will place a call to the createInstance() method of the ObserverCM class. When making this instantiation call, the designer will be required to pass the pattern identifier, the enrolling Task object, and
Listing 4.7: Observer Pattern Example: Task Interface

```java
public interface Task {
    void addListener(TaskListener listener);
    void removeListener(TaskListener listener);
    Collection getListeners();
    void runTask();
    int getProgress();
}
```

the associated SubjectDefs object. After checking that the enrolling player satisfies its initial value constraint, the instantiation method will create a new contract monitor corresponding to the Observer pattern instance being created. Next, the enrolling player will be registered with the monitor to record its enrollment status. Finally, the method will create a new role monitor that monitors the behavior of the Task object enrolling in the Subject role. One possible implementation of SubjectDefs that might be used is presented in Listing 4.8.

**Explanation of Listing 4.8**

- **Lines 4–8** We assume that the set of attached observers directly corresponds to the Collection of TaskListener objects returned by getListeners(). Hence, getSC_Observers() performs a simple type conversion from the Collection returned by getListeners() to the required return type of Set.
public class TaskDefs
implements SubjectDefs
{
    public Set getSC_Observers(Object player)
    {
        Task task = (Task) player;
        return (new HashSet(task.getListeners()));
    }

    public Object getSC_AS(Object player)
    {
        Task task = (Task) player;
        return (new Integer(task.getProgress()));
    }

    public boolean ac_Modified(Object AS1, Object AS2)
    {
        Integer as1 = (Integer) AS1;
        Integer as2 = (Integer) AS2;
        return (!as1.equals(as2));
    }

    public String[] getMethodMap()
    {
        String[] methodMap = { "addListeners", "removeListener" };  
        return (methodMap);
    }
}

Listing 4.8: Observer Pattern Example: TaskDefs Class
• **Lines 10—14** In this scenario, the application-level state of interest is the subject’s completion percentage. Therefore, getSC_AS() encodes the application-level state of the subject as the value returned by getProgress(). We assume that getProgress() returns an integer between 0 and 100.

• **Lines 16—21** Under this implementation of ac_Modified(), the value of the subject is considered to have been modified whenever its completion percentage changes. This has the effect of requiring the subject to notify its attached observers whenever it makes a single percentage point of progress. To reduce the number of notifications, an alternative definition of Modified — and therefore of ac_Modified() — would be selected. For example, the definition of Modified might group consecutive completion percentage values into equivalence classes in blocks of ten percent. That is, from the perspective of Modified, the state of the subject would be considered unchanged as the completion percentage varied from zero to nine, ten to nineteen, twenty to twenty-nine, etc. In effect, this scheme sets milestones at each increment of ten percent. Only when the subject reaches — or passes — one of these milestones would the subject be required to notify its observers.

• **Lines 23—28** The constant array returned by getMethodMap() indicates that addListener() is the method corresponding to Attach(), and removeListener() is the method corresponding to Detach(). Again, the map entry corresponding to Notify() is missing. We will shortly discuss the absence of this entry in more detail.
public class ProgressBarDefs
  implements ObserverDefs
{
  /* ... methods omitted ... */

  public boolean ac_Consistent(Object sAS, Object oAS)
  {
    Integer subAS = (Integer) sAS;
    Integer obsAS = (Integer) oAS;
    return (subAS.equals(obsAS));
  }
}

Listing 4.9: Observer Pattern Example: ProgressBarDefs Class

When each of the user interface objects enrolls in the pattern instance, the designer will place the appropriate enrollment call to the enrollAsObserver() method of the corresponding contract monitor. The method will first check that the appropriate player conditions are satisfied. Next, the player will be registered with the monitor to record its enrollment status. Finally, the method will create a new role monitor to be swapped with the enrolling player. To customize the behavior of the new role monitor to the player being monitored, the designer must supply an object that implements the ObserverDefs interface. The implementation of this object will follow a structure similar to the SubjectDefs implementation presented above. We consider two possible ObserverDefs implementations — one corresponding to each type of observer.

Explanation of Listing 4.9

In Listing 4.9, we present a key portion of the ObserverDefs implementation corresponding to the progress bar object. We assume that the application-level state of the progress bar is encoded as an integer that represents the percentage of the bar rendered by the object. As the progress bar is intended to indicate the progress of
public class ButtonDefs
  implements ObserverDefs
{
  /* ... methods omitted ... */

  public boolean ac_Consistent(Object sAS, Object oAS)
  {
    Integer subAS = (Integer) sAS;
    Boolean obsAS = (Boolean) oAS;
    boolean consistent =
    (((subAS.intValue() == 100) && obsAS.equals(Boolean.TRUE)) ||
     ((subAS.intValue() != 100) && obsAS.equals(Boolean.FALSE)));
    return (consistent);
  }
}

Listing 4.10: Observer Pattern Example: ButtonDefs Class

the running task, the definition of Consistent requires the percentage of the progress
bar that is filled to be identical to the completion percentage of the task. Hence,
ac_Consistent() checks whether the application-level state of the subject is identical to
the application-level state of the observer.

Explanation of Listing 4.10

In Listing 4.10 we present a key portion of the ObserverDefs implementation
corresponding to the button object. In the case of the button, the application-level
state of interest is the value of the button’s enabled property — a property which
indicates whether the button will respond to click events. We therefore assume that
the application-level state of the button object is encoded as a boolean value. Since
the button is provided to view the results of the running task, it should only be
enabled when the task is complete. Hence, the definition of Consistent requires that
the button be enabled if and only if the completion percentage of the task is equal to
one hundred percent. This is the definition of consistency implied by the definition of ac.Consistent().

4.5 Discussion

One of the principal advantages of the monitoring approach is the flexibility that it provides to system designers. In addition to allowing customization based on auxiliary concept definitions, individual monitors make no assumptions about how individual state components are represented. Moreover, monitors make no assumptions about the interfaces exposed by participating objects. Hence, unlike traditional wrapper-based techniques, contract monitors can be used to check the behavior of pattern participants regardless of the interfaces exposed by those participants. Indeed, each monitor provides the same degree of flexibility as the contract that it checks.

Another interesting advantage of the monitoring approach is the ability of a monitor to check requirements imposed on an intentionally defined set of methods. Specifically, a monitor can check that all application-level methods respect a particular requirement without explicitly identifying the methods in this set. This property is especially beneficial when performing routine maintenance activities. Consider, for example, the application-level requirements imposed on subject objects in the Observer pattern. Suppose a designer, as part of maintaining a system, adds a new method to the subject’s class that materially modifies the subject’s state. Even if the designer verifies that the method respects the appropriate class invariants, problems may arise if the designer neglects to call Notify() from within this method, as one or more observers may be left in a state that is inconsistent with the state of the subject. Assuming the system is instrumented with the appropriate monitor, this type
of violation would be detected without any modifications to either the monitor or the system.

The approach is not, however, without its limitations. To understand one of the most important, recall the behavior of the Dynamic Proxy API. In Java, the API creates each proxy based on a set of interfaces to be provided by the proxy. The object created by the API will expose only those methods defined in the corresponding interface modules. This has two important implications for a designer who wishes to use our approach. First, the designer must program to interfaces rather than classes. Otherwise, player methods that are not defined in an interface module will be hidden by the player’s role monitor. As programming to interfaces is considered standard practice in many design disciplines, we see this as only a minor limitation.

Unfortunately, the second implication is more serious. Since interface modules describe only the public methods exposed by an object, the monitoring approach can only detect contract violations involving public methods. Indeed, this was the motivation for excluding the private `Notify()` method in our monitoring example.

At this point, it is worth mentioning that the analogous API in Microsoft’s .NET Framework behaves somewhat differently. In .NET, if an object’s class inherits from a particular supertype, the API will generate a proxy for that object that provides exactly the same methods. The API does not require the “proxied” methods to be defined in interface modules, nor does it require the methods to be declared as public. While this allows us to relax the requirement of programming to interfaces, it does not completely address the issue of monitoring private methods. To understand why the limitation persists, recall that when an object enrolls in a particular role, the corresponding role monitor is swapped with the object and used in its place.
Thereafter, the object must be accessed only through its role monitor. This presents a problem for private methods, since these methods are typically called through a self-reference (e.g., this). Consequently, these calls bypass the checking behavior provided by the role monitor.

One possible solution to this problem is to require the system designer to introduce an auxiliary self-reference within each player object. Upon construction, this self-reference would be set equal to the self-reference provided by the programming language. At the point of enrollment, the contract monitor would update this data member to reference the player through its corresponding role monitor. The designer would be required to place all private method calls through the auxiliary self-reference, rather than through the self-reference provided by the language. While this would address the problem, the solution isn’t particularly appealing, as it requires designers to change their programming style.

There is another limitation which should be mentioned — again associated with the use of checking wrappers. When an object enrolls in a pattern instance, and the designer places the appropriate enrollment call, the perceived identity of the enrolling player changes. That is, the player’s role monitor is not entirely transparent to the client code. If the correctness of the client depends on the identity of the objects with which it participates, a monitored player could present a correctness problem. Consider, for example, the simple case of membership testing based on object identity. If an object is first added to a collection, and is then wrapped by a role monitor, future membership tests will indicate that the player is not contained in the collection.
In the next chapter, we consider an alternative approach to monitoring design pattern contracts. The approach that we consider relies on aspect-oriented programming (AOP) principles [Kiczales et al. 2001]. As we will see, the alternative requires some specialized knowledge, but, in principle, resolves the limitations associated with the wrapper-based approach.

4.6 Chapter Summary

A contract monitor is an executable unit of deployment used to determine whether a design pattern contract is respected at runtime. To monitor a particular pattern instance, the designer is required to create a corresponding contract monitor at the point the instance is created. Thereafter, the designer is required to place appropriate calls to the monitor whenever an object enrolls or disenrolls from the pattern instance.

When a player enrolls in a pattern instance, the corresponding enrollment call creates a role monitor customized to the enrolling player. This object exposes the same interface as the player, and is used by clients in its place. When a call is dispatched through the role monitor, the monitor checks that both the caller and the callee satisfy the appropriate role-level requirements. If a contract violation is detected, an appropriate error message is raised; otherwise, the call is delegated to the underlying player object.

Every contract monitor provides the same degree of flexibility as the contract that it checks. This flexibility is achieved by parameterizing the monitor with respect to the same abstraction functions, auxiliary concepts, and method mappings as the corresponding contract. By supplying the appropriate executable definitions for these
notions, the designer can tailor the monitor’s behavior to the pattern instance in question.
“It is what I sometimes have called 'the separation of concerns', which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts, that I know of.”

— Edsger W. Dijkstra.

5.1 Chapter Overview

In the previous chapter, we described an approach to monitoring whether the contracts used in developing a system are respected at runtime. In this chapter, we present an alternative monitoring technique based on the use of aspect-oriented programming (AOP) [Kiczales et al. 1997]. As we will see, this approach offers a number of advantages. We begin by presenting an overview of AOP with AspectJ [Kiczales et al. 2001] — an implementation technology for Java that provides the foundation for our discussion. We then present the basic monitoring approach, and illustrate the approach in the context of an example. We conclude with a discussion of the advantages and disadvantages associated with the AOP-based monitoring approach.
5.2 Aspect-Oriented Programming with AspectJ

Aspect-oriented programming (AOP) is a technique for achieving a clear separation of concerns in object-oriented software. The fundamental claim driving the approach is that existing object-oriented decomposition mechanisms are insufficient in simultaneously modularizing the relevant concerns in complex software. In the same spirit as [Tarr, Ossher, Harrison and Sutton, Jr. 1999], proponents of AOP argue that the decomposition boundaries required to modularize some concerns preclude the modularization of others. That is, given one decomposition of a system into classes, some concerns will inherently cross-cut class boundaries. The code associated with these cross-cutting concerns will be tangled throughout the system’s implementation.

AOP enables the modularization of cross-cutting concerns through a code factorization strategy. To understand the approach, consider a system decomposed into classes, and a particular concern that cross-cuts the boundaries of those classes. Now imagine marking within each class the code fragments associated with the cross-cutting concern, and assigning a label to each unique code fragment. Next, imagine factoring those fragments into a separate compilation unit, and replacing each fragment within the classes with the corresponding fragment label. The new compilation unit contains only the code relevant to a single area of concern, and provides enough information for a sophisticated compiler to weave the code fragments back into their appropriate locations. This essentially describes the AOP process in reverse. AOP languages provide linguistic support for identifying particular points within a system’s
execution where additional code fragments should be executed\textsuperscript{35}. The modules that contain these fragments and identify the corresponding insertion points are referred to as “aspects”. These aspects are developed as separate modules — not developed as part of other classes and then factored out — and are \textit{woven} into the final system code using a specialized compiler.

AspectJ is an AOP implementation for Java, and has become the defacto standard for AOP-based development. Aspect modules in AspectJ leverage a number of new language concepts. We focus on three of the most important: (i) join points, (ii) pointcuts, and (iii) advice. A \textit{join point} identifies a unique point in a system’s execution, such as when a particular method is called, or when a particular constructor is executed. Each join point corresponds to a point of insertion where additional code will be inserted by the weaving compiler. A \textit{pointcut} is a set of join points that should be treated uniformly. For example, whenever a call is placed to \textit{any} method of a particular class, it might be appropriate to execute the same code fragment. The AspectJ language extensions provide primitive \textit{pointcut designators} that identify simple sets of join points. These designators can be composed to identify more complex join point sets. Moreover, these designators can collect information about the execution context that can be passed to the code inserted at each join point. For example, a designator might be used to collect the object on which a method was called, or the arguments that were passed to the method. Finally, associated with each pointcut is an \textit{advice} block that specifies the code to be executed whenever control reaches one

\textsuperscript{35}In our example, we considered execution points based on the syntactic structure of the underlying modules. In general, however, AOP allows for the identification of execution points based on runtime phenomenon such as state conditions and call sequence information.
of the join points within the set. There are three forms of advice to consider: (i) before
advice, (ii) after advice, and (iii) around advice. To understand the distinction
among the three, consider a join point that matches a method invocation. Before
advice is executed before the method is executed, and after advice is executed after
the method is executed. Around advice surrounds the invocation, and controls when
the invocation proceeds. This type of advice can be used to add both pre-processing
and post-processing behavior to the matching invocation.

It is worth mentioning that aspect implementations in AspectJ are similar to
standard class implementations in Java. Just like a class, an aspect can inherit from
a supertype, and may be declared as abstract. Pointcuts defined by an aspect may
also be declared as abstract, and support the standard accessibility modifiers (i.e.,
public, private, and protected). Moreover, an aspect can define fields and methods
to support its task. Again, the standard accessibility modifiers are supported, and
method implementations may be deferred to subaspects. The similarity between
a class and an aspect will be evident when we turn our attention to the example
presented in Section 5.4.

One important difference that we should mention at this point, however, is that
aspects are typically “singletons”. That is, for each aspect implementation, only
one instance of that implementation is created per system run. Each singleton is
created automatically by the runtime during the system’s initialization. While As-
pectJ supports additional aspect creation strategies, the singleton strategy is the most
commonly used, and remains our focus hereafter.
5.3 The Monitoring Approach

A design pattern is intended to describe a behavioral and structural slice through a system. Indeed, this cross-cutting structure is an important part of what makes the pattern a good solution in the eyes of the object-oriented community. As a consequence of this structure, the proof obligations that must be dispatched when showing that a pattern has been applied correctly are also inherently cross-cutting. In designing a runtime monitoring system that checks whether these obligations are satisfied, an AOP-based approach is therefore a natural choice. After all, the whole point of AOP is to provide linguistic support for modularizing concerns — such as contract monitoring — that span class boundaries.

In the AOP-based approach, we again focus on the design of contract monitors that can be deployed on a per contract basis. In our approach, each contract monitor is implemented as an abstract aspect. The code of the abstract aspect provides the monitoring functionality common across all instances of the pattern to which it corresponds. The parts that are variable among different applications of the pattern are deferred to a subaspect that appropriately customizes the monitor’s behavior. The subaspect and the abstract aspect together form the complete monitoring code for one or more pattern instances.

Recall that in the wrapper-based monitoring approach, the designer is required to supply executable definitions for the appropriate abstraction functions and auxiliary concepts. The definitions supplied by the designer are used to tailor the behavior of the monitor to a particular pattern application. We take a similar approach in designing our aspect-based monitors. Specifically, each abstraction function and auxiliary concept is represented by an abstract method within the abstract aspect. The
aspect’s monitoring code is expressed over these methods — definitions of which are supplied by the subaspect appropriate to one or more pattern instances. Note that since the method definitions are supplied per subaspect, and not per player, some methods will select their behavior based on runtime type information and/or object identity. For example, consider the subaspect corresponding to an instance of the Observer pattern. If multiple observers participate in an instance of the pattern, the method corresponding to the Consistent auxiliary concept will determine its notion of consistency based on the type and/or identity of the observer in question.

Next, recall that in the wrapper-based approach, the designer is additionally required to supply appropriate method and parameter mappings. These mappings are used by the monitor to determine which assertions to check for each player method. In the aspect-based approach, this information is provided using AspectJ’s pointcut construct. The pointcuts defined in the abstract aspect correspond to the methods defined in the associated role-level specifications. The advice associated with a particular pointcut provides the checking behavior appropriate to the corresponding method. Each pointcut is declared as abstract, allowing the designer to supply a definition that binds the checking advice to the appropriate player method. Moreover, each abstract pointcut defines the parameters that must be collected by definitions of that pointcut, allowing designers to specify the appropriate parameter mappings. Each parameter defined in an abstract pointcut corresponds to a method parameter defined in the role-level specification. Again, these are the parameters material to the assertions which must be checked by the monitor’s advice. When a designer provides a pointcut definition, the parameters collected will be passed to the advice associated with that pointcut.
Note that if a pattern contract imposes application-level method requirements on the objects playing a particular role, the corresponding monitor will introduce an additional pointcut to capture these methods. The advice associated with this pointcut will check that the appropriate application-level method requirements are respected. In a subaspect, the designer must provide a definition that matches invocations on any player method that does not correspond to one of those defined by the role-level specification. As we will soon see, the pointcut construct provides a convenient mechanism for binding checking behavior to an intentionally defined set of methods.

Next, recall that in the wrapper-based approach, the designer is required to make calls to the monitoring library at specific points in the system’s execution. Specifically, the designer is required to place library calls whenever a pattern instance is created, and whenever an object enrolls or disenrolls from a pattern instance. Again, these are the points where verification conditions materialize in the program code. However, in the aspect-based approach, these calls are not required. Instead, the monitor again employs the use of abstract pointcuts to flexibly capture these execution points. So, for example, a monitor may define a pointcut corresponding to the point of pattern creation. In the subaspect associated with a particular pattern instance, the designer will then provide a definition of that pointcut that binds the point of pattern instantiation to the appropriate execution point. In general, a monitor may define pointcuts corresponding to the point of pattern instantiation, as well as the points of player enrollment and disenrollment. Typically, however, the monitor will combine some pointcuts for convenience. For example, in the Observer pattern monitor presented in Section 5.4, the pointcut corresponding to the point of observer enrollment is the same pointcut corresponding to the observer’s Attach() method.
Finally, recall that a monitor may have to introduce additional state components to check whether a method satisfies its call sequence requirements. Typically, these state components are associated with individual player objects. In the wrapper-based approach, the components associated with a particular player are introduced in the player’s invocation handler. In the aspect-based approach, however, all of the player objects participating in a pattern instance are monitored by the same aspect instance. Therefore, additional state components are stored within a single aspect. The aspect maintains the associations between player objects and their respective state components. We will see an example of this structure when we consider the Observer pattern monitor in the following section.

Before turning to the example, however, it is worth mentioning that AspectJ provides an alternative technique for storing auxiliary player state. To this point in the discussion, we have focused on AspectJ’s *dynamic* cross-cutting features. These features provide the ability to define additional computation at well-defined execution points. The language additionally provides support for *static* cross-cutting. These features make it possible to modify the static type signatures of a system’s classes. These static cross-cutting features could be used to add the appropriate auxiliary state components and accessor methods to a player’s class, rather than storing the information within the aspect. Since storing the state within the aspect allows for a more straightforward exposition, this is the approach that we present in the remainder of the chapter. We do, however, briefly consider the alternative implementation strategy later in our discussion.
5.4 Example: The Observer Pattern

To illustrate the aspect-based monitoring approach, we now turn our attention to an example. We once again consider the monitor corresponding to the Observer pattern contract presented in Chapter 3. Section 5.4.1 presents the abstract aspect that implements the monitoring behavior common across all instances of the Observer pattern. Section 5.4.2 describes a particular application of the pattern, and presents the corresponding subaspect used to customize the monitor’s behavior.

Note that the abstract aspect presented in the following sections is designed to support multiple pattern instances. However, the implementation requires that the enrollment sets of different instances be disjoint. That is, the monitor implementation assumes that player objects participating in one instance of the Observer pattern do not participate in any other instance of the Observer pattern. After presenting the example, we will discuss this limitation in detail.

5.4.1 Observer Pattern Contract Monitor

Key portions of the abstract aspect associated with the Observer pattern are presented in the following three listings. Listing 5.1 presents the state components, abstract methods, and abstract pointcuts defined by the aspect. Listing 5.2 presents the advice used to check the behavior of a subject’s Notify() method. Finally, Listing 5.3 presents the advice used to check the behavior of an observer’s application-level methods. Following the format of previous chapters, the code fragments are explained in the subsections following each listing.
public abstract aspect ObserverCM
{
    protected Set enrolledSubjects = new HashSet();
    protected Set enrolledObservers = new HashSet();
    protected Map sub_lastNotifiedAS = new HashMap();
    protected Map obs_lastSubjectAS = new HashMap();
    protected Map obs_updateCalled = new HashMap();
    /* ... fields omitted ... */

    abstract protected Set sub_getSC_Observers(Object subject);
    abstract protected Object sub_getSC_AS(Object subject);
    abstract protected
        boolean sub_ac_Modified(Object sub_AS1, Object sub_AS2);

    abstract protected Object obs_getSC_Subject(Object observer);
    abstract protected Object obs_getSC_AS(Object observer);
    abstract protected
        boolean obs_ac_Consistent(Object sub_AS, Object obs_AS);

    abstract protected pointcut subjectEnroll(Object subject);
    abstract protected pointcut subjectDisenroll(Object subject);

    abstract protected
        pointcut attach(Object subject, Object observer);
    abstract protected
        pointcut detach(Object subject, Object observer);
    abstract protected pointcut notify(Object subject);
    abstract protected pointcut update(Object observer);

    abstract protected pointcut subjectOthers(Object subject);
    abstract protected pointcut observerOthers(Object observer);

    /* ... advice omitted ... */
}
Explanation of Listing 5.1

- **Lines 3–8** The ObserverCM aspect declares a number of data members to support its monitoring task. The first two members, `enrolledSubjects` and `enrolledObservers`, are used to maintain the Subject and Observer enrollment sets, respectively. When an object enrolls in the Subject role, the advice corresponding to the point of enrollment adds the enrolling player to `enrolledSubjects`. When a subject disenrolls, the corresponding advice removes the player from this Set. The `enrolledObservers` Set is used similarly for observer objects. Both data members are used by the monitor in checking whether player enrollment conditions are satisfied. For example, when a player object tries to enroll in the Observer role, the corresponding advice checks that the player is not currently enrolled as either a Subject or an Observer.

The aspect next defines three data members used to associate auxiliary state components with the players participating in an instance of the Observer pattern. These state components parallel those maintained by the invocation handlers discussed in the previous chapter. The first of these three, `sub_lastNotifiedAS`, is used to record the application-level state of a subject at the point `Notify()` was last called on that subject. The second, `obs_lastSubjectAS`, is used to record for each observer, the value of its subject at the point the observer was last accessed. Finally, `obs_updateCalled` associates a boolean flag with each observer that tracks whether `Update()` was called on the observer since the last time the flag was cleared by the monitor. We will discuss how these data members are used by the monitor in detail when we turn our attention to the checking advice implemented by the aspect.
• Lines 10–18. ObserverCM next defines the abstract methods that must be implemented in a subaspect to customize the monitor’s behavior to a particular application. These methods directly parallel those defined in the SubjectDefs and ObserverDefs interfaces discussed in the previous chapter. Indeed, as we will soon see, implementations of these methods are also similar to those discussed previously. The first three methods are associated with objects playing the Subject role. The second three are associated with objects playing the Observer role. As in the wrapper-based approach, these methods correspond to the abstraction functions and auxiliary concepts associated with objects playing these roles. For a review of these methods, and their relationship to the Observer pattern contract, the reader is referred to Chapter 4, Section 4.4.1.

• Lines 20–31. The abstract aspect next defines the pointcuts that must be implemented in a subaspect to bind the monitor’s checking advice to the appropriate system execution points. The first two pointcuts, subjectEnroll and subjectDisenroll, represent the points where a subject object enrolls or disenrolls, respectively, from an instance of the Observer pattern. Recall that these points coincide with the points of pattern creation and destruction, respectively. The pointcuts define the subject parameter, which captures the player object enrolling or disenrolling. When implemented in a subaspect, the actual pointcut must bind the appropriate subject object to this parameter, which will then be passed to the advice associated with the pointcut. Notice that the aspect does not define the analogous pointcuts corresponding to the Observer role. For convenience, these execution points are captured by the pointcuts corresponding to the Attach() and Detach() methods. More precisely, when an object attaches
to its *subject*, it is automatically assumed to have enrolled in the *Observer* role. Similarly, when an object detaches from its *subject*, it is automatically assumed to have disenrolled from the *Observer* role. Hence, the checking advice corresponding to these points is bound to the pointcuts used to capture the *Attach()* and *Detach()* methods.

The *attach*, *detach*, and *notify* pointcuts correspond to the *subject* methods that satisfy the *Attach()*, *Detach()*, and *Notify()* method requirements. The first parameter to each of these pointcuts captures the target of the method invocation. The second parameter to *attach* and *detach* captures the *observer* object that is attaching or detaching. Similarly, the *update* pointcut corresponds to the *observer* method that satisfies the *Update()* method requirement. The single parameter defined by this pointcut is used to capture the target of the invocation.

Finally, the last two pointcuts are used to bind checking advice to the application-level methods of objects playing the *Subject* and *Observer* roles. As the name suggests, *subjectOthers* corresponds to all of the methods provided by an object playing the role of *Subject*, other than those corresponding to methods defined by the role-level specification. The *observerOthers* pointcut is defined analogously.

We next focus on the advice used by the aspect to check whether contract requirements are satisfied. We present the advice associated with the *notify* and *observerOthers* pointcuts. The advice associated with other pointcuts has been omitted, as it follows a nearly identical pattern.
Object around(Object subject): notify(subject)
{
    if(!enrolledSubjects.contains(subject))
    { return(proceed(subject)); }  

    Set observers_PRE = sub_getSC_Observers(subject);
    Object subAS_PRE = sub_getSC_AS(subject);

    Iterator obs = observers_PRE.iterator();
    while(obs.hasNext())
    { obs_updateCalled.put(obs.next(), Boolean.FALSE); }

    sub_lastNotifiedAS.put(subject, subAS_PRE);
    Object result = proceed(subject);

    Set observers_POST = sub_getSC_Observers(subject);
    Object subAS_POST = sub_getSC_AS(subject);

    if(!subAS_PRE.equals(subAS_POST))
    { violation("! subject not preserved"); }

    if((observers_PRE.size() != observers_POST.size()) ||
        (!observers_PRE.containsAll(observers_POST)))
    { violation("! observers set not preserved."); }

    Iterator obs = observers_POST.iterator();
    while(obs.hasNext())
    {
        Object observer = obs.next();
        Boolean updateCalled = (Boolean) obs_updateCalled.get(observer);
        Object lastSubjectAS = obs_lastSubjectAS.get(observer);
        
        if(updateCalled.equals(Boolean.FALSE))
        { violation("! observer not updated."); }

        if(sub_ac_Modified(subAS_POST, lastSubjectAS))
        { violation("! subject preservation violation."); }
    }

    return(result);
}

Listing 5.2: Observer Contract Monitor: ObserverCM Aspect (2 of 3)
Explanation of Listing 5.2

The advice bound to the notify pointcut checks the behavior of the player method corresponding to Notify(). The advice first checks whether the target object is enrolled in the Subject role. If not, the advice is skipped, and control is passed to the Notify() method. After all, if the object isn’t enrolled in a pattern instance, there are no role-level requirements to check. Next, the advice takes a snapshot of the pre-conditional values required to check the player method’s post-condition.

Before executing the actual player method, the advice updates the auxiliary state components required to check the call sequence conditions imposed by the pattern contract. First, the advice clears the updateCalled flag associated with each observer attached to the subject. The advice corresponding to the Update() method sets the flag associated with the target of the invocation. As we will see, this allows the notify advice to check that Update() was called on each attached observer. Second, the advice records the application-level state of the subject at the point Notify() was called. This auxiliary state component is used by the advice associated with application-level subject methods to check that these methods appropriately invoke Notify() when the state of the subject is materially modified.

After invoking the actual player method corresponding to Notify(), the advice checks that the method’s post-condition was satisfied. First, the advice checks that the application-level state of the subject was preserved. The advice then checks that the player method preserved the set of attached observers. Finally, the advice checks that the appropriate call sequence conditions were satisfied. Specifically, the advice

36 AspectJ’s proceed() construct is used to pass control from around advice to the join point that it surrounds.
checks that each observer was updated during the call, and that the preservation condition imposed on subject methods was respected. That is, using the lastSubjectAS value associated with each observer, the advice checks that the post-conditional value of the subject is unmodified — according to the definition of Modified — with respect to its value at the point of the last call to each observer. If a violation is detected at any point, the advice displays the appropriate error message and terminates the system.

Explanation of Listing 5.3

The advice bound to the observerOthers pointcut checks the behavior of application-level methods provided by objects playing the Observer role. Again, the advice begins by checking whether the target of the invocation is an enrolled player, and if not, allows the invocation to proceed unchecked. Otherwise, the advice records the pre-conditional values required to check that the method satisfies the relevant post-conditions, and records the application-level state of the observer’s subject before allowing the call to proceed. As discussed in the description of the advice associated with the notify pointcut, the value of the subject recorded at this point is used to check — in the advice associated with subject methods — whether the post-conditional value of the subject is unmodified with respect to its value at the point of the last call to this observer. Finally, the advice allows the invocation to proceed, and checks that the appropriate post-conditions are satisfied. Specifically, the advice checks that the

37Eliminating this somewhat paranoid check results in a simplified aspect implementation — beyond the simplification of this particular advice block alone. We include the check, however, to remain consistent with the contract presented previously. The reader is referred to Chapter 3 Section 3.7.1 for a review of this subtle preservation requirement.
/* ... advice omitted ... */

Object around(Object observer): observerOthers(observer)
{
    if(!enrolledObservers.contains(observer))
    { return proceed(observer); }

    Object sub_PRE = obs_getSC_Subject(observer);
    Object sub_AS_PRE = sub_getSC_AS(sub_PRE);

    obs_lastSubjectAS.put(observer, sub_AS_PRE);

    Object result = proceed(observer);

    Object sub_POST = obs_getSC_Subject(observer);
    Object sub_AS_POST = sub_getSC_AS(sub_POST);
    Object obs_AS_POST = obs_getSC_AS(observer);

    if(sub_PRE != sub_POST)
    { violation("! subject not preserved."); } }

    if(!obs_ac_Consistent(sub_AS_POST, obs_AS_POST))
    { violation("! inconsistent observer."); }

    return(result);
}

private void violation(String message)
{ /* ... body omitted ... */ }

Listing 5.3: Observer Contract Monitor: ObserverCM Aspect (3 of 3)
observer’s subject reference was preserved, and that the final state of the observer is consistent — according to the definition of Consistent — with the state of its subject.

5.4.2 Sample Application

To demonstrate the application of the aspect-based monitor, we again consider the Observer pattern applied in the context of designing user interfaces. In particular, we focus on the scenario presented in Chapter 4. In that example, the Subject role is played by an object that encapsulates the code required to complete some long running task. We again assume that the object implements the Task interface, and in this case, performs a file manipulation task. The key portions of the subject’s class implementation are presented in Listing 5.4. Again, the Observer role is played by two user interface objects, each of which implements the TaskListener interface. The first type of observer renders a linear bar graph used to signal the user of the task’s progress. The key portions of this observer’s class implementation are presented in Listing 5.5. The second type of observer renders a button used to display the results of the file manipulation task. The key portions of this observer’s class implementation are presented in Listing 5.6.

We now consider the subaspect used to customize the behavior of the monitor to our sample application. The subaspect implementation is presented in Listings 5.7 and 5.8.

Explanation of Listing 5.7

The subaspect implementation begins by providing definitions for each of the abstract pointcuts defined in the ObserverCM aspect. The first three pointcut definitions correspond to the subject’s Attach(), Detach(), and Notify() methods. Therefore, the
public class IOTask implements Task {
    /* ... fields omitted ... */
    public void addListener(TaskListener listener) {
        /* ... body omitted ... */
    }
    public void removeListener(TaskListener listener) {
        /* ... body omitted ... */
    }
    public Collection getListeners() {
        /* ... body omitted ... */
    }
    public void runTask() {
        /* ... body omitted ... */
    }
    public int getProgress() {
        /* ... body omitted ... */
    }
    private void notifyObs() {
        /* ... body omitted ... */
    }
    /* ... methods omitted ... */
}

Listing 5.4: Observer Pattern Example: IOTask Class
public class TaskProgressBar implements TaskListener {
    /* ... fields omitted ... */

    public TaskProgressBar(Task task) {
        /* ... body omitted ... */
    }

    public void update() {
        /* ... body omitted ... */
    }

    public Task getTask() {
        /* ... body omitted ... */
    }

    public int getValue() {
        /* ... body omitted ... */
    }

    /* ... methods omitted ... */
}

Listing 5.5: Observer Pattern Example: TaskProgressBar Class

public class TaskOutputButton implements TaskListener {
    /* ... fields omitted ... */

    public TaskOutputButton(Task task) {
        /* ... body omitted ... */
    }

    public void update() {
        /* ... body omitted ... */
    }

    public Task getTask() {
        /* ... body omitted ... */
    }

    public boolean isEnabled() {
        /* ... body omitted ... */
    }

    /* ... methods omitted ... */
}

Listing 5.6: Observer Pattern Example: TaskOutputButton Class
public aspect IOTaskCM
    extends ObserverCM
{
    protected pointcut attach(Object subject, Object observer)
        :(execution(void IOTask.addListener(TaskListener)) &&
         target(subject) && args(observer));

    protected pointcut detach(Object subject, Object observer)
        :(execution(void IOTask.removeListener(TaskListener)) &&
         target(subject) && args(observer));

    protected pointcut notify(Object subject)
        :(execution(void IOTask.notifyObs()) && target(subject));

    protected pointcut update(Object observer)
        :(execution(void TaskListener+.update()) && target(observer));

    protected pointcut subjectEnroll(Object subject)
        :(execution(IOTask.new(..)) && target(subject));

    protected pointcut subjectDisenroll(Object subject)
        :(execution(void IOTask.finalize()) && target(subject));

    protected pointcut subjectOthers(Object subject)
        :(execution(* IOTask.*(..)) && target(subject) &&
          !execution(void IOTask.finalize()) &&
          !execution(void IOTask.addListener(TaskListener)) &&
          !execution(void IOTask.removeListener(TaskListener)) &&
          !execution(void IOTask.notifyObs()));

    protected pointcut observerOthers(Object observer)
        :(execution(* TaskListener+.*(..)) && target(observer) &&
          !execution(void TaskListener+.finalize()) &&
          !execution(void TaskListener+.update()));
}

Listing 5.7: Observer Pattern Example: IOTaskCM Aspect (1 of 2)
pointcut definitions capture the appropriate methods implemented by IOTask. In each case, AspectJ’s target() construct is used to collect the target of the method invocation. Similarly, in the case of attach and detach, AspectJ’s args() construct is used to collect the observer object passed as argument. The fourth pointcut corresponds to the observer’s Update() method, and is defined similarly. The key syntactic difference between this definition and the first three is the addition of the “+” following TaskListener. This symbol indicates that the pointcut captures the execution of update() in any class that implements the TaskListener interface or some interface that extends it.

The next two pointcut definitions correspond to the points where a subject object enrolls or disenrolls, respectively. In the sample application, we assume that the subject object enrolls immediately upon construction, and disenrolls just before going out of scope. Therefore, the subjectEnroll pointcut captures the creation of IOTask objects, and subjectDisenroll captures the execution of the finalize() method on these objects.

Finally, the subjectOthers and observerOthers pointcuts capture the application-level methods of objects playing the Subject and Observer roles, respectively. Hence, subjectOthers captures all of the IOTask methods other than those used to satisfy role-level method requirements. The pointcut additionally excludes the finalize() method, since this method is executed only after an object goes out of scope. The observerOthers pointcut is defined analogously.

38Note that finalize() is the method executed by Java’s garbage collector just before the memory used by an object is reclaimed.
```java
protected Set sub_getSC_Observers(Object subject) {
    /* ... body omitted ... */
}

protected Object sub_getSC_AS(Object subject) {
    /* ... body omitted ... */
}

protected boolean sub_ac_Modified(Object sub_AS1, Object sub_AS2) {
    /* ... body omitted ... */
}

protected Object obs_getSC_Subject(Object observer) {
    /* ... body omitted ... */
}

protected Object obs_getSC_AS(Object observer) {
    if (observer instanceof TaskOutputButton) {
        TaskOutputButton button = (TaskOutputButton) observer;
        return (new Boolean(button.isEnabled()));
    }
    else {
        TaskProgressBar bar = (TaskProgressBar) observer;
        return (new Integer(bar.getValue()));
    }
}

protected boolean obs_ac_Consistent(Object sub_AS, Object obs_AS) {
    Integer subAS = (Integer) sub_AS;
    if (obs_AS instanceof Integer) {
        Integer obsAS = (Integer) obs_AS;
        return (sub_AS.equals(obsAS));
    }
    Boolean obsAS = (Boolean) obs_AS;
    boolean consistent
        = (((subAS.intValue() != 100) && (!obsAS.booleanValue())) ||
            ((subAS.intValue() == 100) && (obsAS.booleanValue())));
    return (consistent);
}
```

Listing 5.8: Observer Pattern Example: IOTaskCM Aspect (2 of 2)
Explanation of Listing 5.8

The subaspect next provides implementations of the abstract methods defined by ObserverCM. Again, these methods directly parallel those defined by the SubjectDefs and ObserverDefs interfaces discussed in the previous chapter. Indeed, the implementations of these methods are also similar to those used in the wrapper-based approach. The primary distinction is that the subaspect implements the methods used across all pattern participants, whereas an implementation of SubjectDefs or ObserverDefs implements the methods specific to a particular pattern participant. Hence, some method implementations in the subaspect must select their behavior based on the type and/or identity of the objects passed as argument.

Consider, for example, the obs_getSC_AS() method, which is used by the monitor to produce an encoding of an observer’s application-level state. Since this method is used across all observers, it must select its behavior based on the type of observer passed as argument. If the observer in question is an instance of TaskOutputButton, the method encodes the application-level state of the observer as a boolean value that indicates whether the observer is enabled. Otherwise, the observer must be an instance of TaskProgressBar, and the method encodes the observer’s application-level state as the percentage of the bar that is currently filled.

Similarly, consider the implementation of obs_ac_Consistent(), which is used by the monitor to determine whether the state of an observer is consistent with the state of a subject. If the application-level state of the observer is encoded as an integer, the method applies the notion of consistency appropriate to an observer of type TaskProgressBar. Otherwise, the application-level observer state must encode a...
boolean value, and the method applies the notion of consistency appropriate to an observer of type TaskOutputButton.

Correcting a Subtle Implementation Error

In light of the pointcut definitions and method implementations presented in Listings 5.7 and 5.8, a careful examination of the checking advice in Listings 5.2 and 5.3 reveals a subtle problem. To see the problem, consider invoking the \texttt{getValue()} method on the progress bar object to determine the percentage of the bar that is currently filled. As this method corresponds to an application-level method, the around advice presented in Listing 5.3 will be executed. After invoking the actual \texttt{getValue()} method on the observer, the advice will check that the observer is consistent with its subject — according to the definition of \texttt{obs.ac.Consistent()}. As part of calling this method, the advice must retrieve an encoding of the post-conditional value of the observer’s application-level state using \texttt{obs.getSC_AS()}. To produce this encoding, the method will invoke \texttt{getValue()} on the player, which will once again cause the advice in Listing 5.3 to begin executing. Eventually, the unbounded recursion will cause the stack to overflow, and the system will terminate.

The key point is that checking advice may need to invoke player methods to which checking advice is bound. To prevent the type of unbounded recursion characterized by the \texttt{getValue()} example, advice must be dynamically enabled and disabled. Player methods invoked as part of advice execution should not be subject to checking. To achieve this effect, every abstract aspect declares a boolean flag \texttt{inAdvice} used to record whether or not advice is currently executing. Whenever control reaches an advice block, the advice checks if the \texttt{inAdvice} flag is set to \texttt{true}. If so, the advice is “short-circuited” and control is passed to the appropriate join point. Otherwise,
the flag is set to `true`, and the advice proceeds as usual. Before control leaves the advice — either normally or through a call to `proceed()` — the flag is again reset to `false`. The approach essentially allows player methods to be invoked unchecked while control is within advice. This implementation subtlety was ignored in the example, so as to better convey the overall approach.

### Overlapping Enrollment Sets

Recall that the implementation of `ObserverCM` requires that player enrollment sets be disjoint across pattern instances. That is, the monitor implementation requires that objects participating in one instance of the Observer pattern do not participate in any other instance of the Observer pattern. The data structures used by this aspect are simply not rich enough to support overlapping enrollment sets. Take, for instance, the `obs_updateCalled` data member. Recall that this data member is used by the advice associated with the `Attach()` and `Notify()` methods to check that these methods invoke `Update()` on the appropriate observers. To understand the limitation of the aspect implementation, consider an observer `o1` attached to a subject `s1`. Now consider if `o1` were to attach to another subject `s2`, and that during the execution of `s2`'s `Attach()` method, `s2` made a change to `s1` that resulted in `s1` notifying its observers. Since the advice bound to `o1`'s `Update()` method would mark `o1` as having been updated, a failure of `s2` to call `Update()` on `o1` would not be detected. The key point is that `obs_updateCalled` records whether `Update()` was called on each observer by some subject. The data member does not capture whether `Update()` was called by

---

39 Similar implementation strategies will be required to dynamically enable and disable advice when developing other types of AOP applications. Specifically, such strategies will be required whenever advice implementations provide their behavior by invoking methods to which advice might later be bound. Without such strategies, unanticipated pointcut definitions could result in unbounded recursion.
a particular subject. Similar problems are associated with the other data members declared by ObserverCM.

Of course, the data structures used by ObserverCM could be modified to accommodate overlapping enrollment sets, or the static cross-cutting features of AspectJ could be used to introduce the state components into the appropriate classes. Unfortunately, neither solution would resolve the limitation. The methods defined by ObserverCM support the monitoring of multiple pattern instances. Hence, when invoked by advice, the behavior of a particular method depends on the pattern instance being monitored at that point. However, these methods do not receive explicit information identifying the pattern instance being monitored. Instead, method implementations provided in a subaspect assume that each object participates in exactly one pattern instance. Therefore, each method selects its behavior based solely on object type and identity information. Consider, for example, the implementation of obs.get_SC_Suject(), used to retrieve a reference to a particular observer’s subject. When each observer participates in only a single pattern instance, an implementation of the method can determine the subject reference to return based on the observer passed as argument. If, however, a single observer were allowed to participate in multiple pattern instances, how would the method determine which subject reference to return?

Supporting overlapping enrollment sets has additional consequences for checking advice. When each object is restricted to enrolling in only a single pattern instance, the correctness of an invocation on a player method need only be evaluated with respect to the single instance in which the player participates. If a player can enroll in multiple instances, however, the correctness of a single invocation may need to be evaluated with respect to multiple pattern instances. Consider, for example, the
requirement that application-level observer methods leave the state of the observer in a state that is consistent with the subject. If the observer participates in multiple instances of the Observer pattern, the checking advice associated with the observer’s application-level methods must check that the observer is consistent with all of its subjects at the end of each invocation.

In those cases where player objects must participate in multiple instances of the Observer pattern, the solution is to introduce multiple subaspect implementations, each of which operates independently of the others. That is, a particular subaspect of ObserverCM would implement the monitoring behavior appropriate to some group of pattern instances — given the condition that those instances do not have overlapping enrollment sets. Additional subaspects — subject to the same enrollment constraint — would be implemented to monitor the remaining pattern instances. In the worst case, each subaspect of ObserverCM would monitor only a single instance of the Observer pattern. In practice, however, such cases are uncommon.

5.5 Discussion

Like the wrapper-based approach discussed in the previous chapter, the principal advantage of aspect-based monitoring is the flexibility that it provides to practitioners. In this respect, the two approaches share a number of points in common. First, both allow for the customization of auxiliary concept definitions to tailor a monitor’s behavior to a particular pattern instance. Second, monitors developed using either approach make no assumptions about how individual state components are represented, or about the interfaces exposed by pattern participants. As a result, monitors developed using either approach can be used to monitor player objects regardless of
how those objects are implemented. Finally, both approaches make it possible to
design monitors that capture requirements imposed on an intentionally defined set of
methods. As discussed in the previous chapter, this property is especially useful in
detecting contract violations introduced during software maintenance activities.

The two approaches share a number of points in common, and provide simi-
lar advantages. However, aspect-based monitors improve upon their wrapper-based
counterparts in a number of significant ways. Most important, aspect-based monitors
do not suffer from the wrapper-based limitations outlined in the previous chapter.
Recall that the primary limitation of the wrapper-based approach is its inability to
monitor the behavior of private methods. Since the pointcut mechanism of AspectJ
allows advice to be bound to any method — including those marked as private —
this limitation does not exist when using aspects. Also recall that in the wrapper-
based approach, the perceived identity of an object is altered at the point it enrolls
or disenrolls from a pattern instance. This of course has important consequences if
the object’s clients depend on this identity. In the aspect-based approach, the compi-
er weaves the checking advice through an existing set of classes. Since there is no
dynamic code generation or wrapping, the approach does not affect the identity of
checked objects. Moreover, unlike the wrapper-based approach, there is no risk of an
object reference escaping, allowing clients to bypass a monitor.

Another advantage of the aspect-based approach is that new monitors tend to re-
quire less development effort as compared to monitors developed using the wrapper-
based technique. Using the latter approach, the designer is required to implement
classes corresponding to (i) the contract monitor, (ii) the contract monitor factory[40]

[40]As discussed in Chapter 4, the contract monitor class may serve as a factory that creates
instances of itself.
and (iii) the invocation handler associated with each role. In the aspect-based approach, the designer need only implement a single aspect. The aspect generally requires less code than that of the wrapper-based approach. More important, experience working with both monitoring techniques suggests that the aspect code is less complex with respect to human understandability. An important factor in this complexity difference is that checking advice is written at the same level as system code.

In the wrapper-based approach, the checking code is developed at a meta-level. That is, each invocation handler receives a marshalled representation of an invocation, as well as the arguments to that invocation. The designer is required to unmarshall each invocation request, and must provide the appropriate dispatching logic. Experience suggests that these steps are prone to error; they are not required when developing aspect-based monitors.

The aspect-based approach is not, however, without its disadvantages. In particular, it is worth mentioning that while AspectJ’s pointcut construct provides a powerful mechanism for binding checking advice to the appropriate execution points, it is also inherently prone to error. When providing pointcut definitions, it is difficult for a designer to determine if the pointcuts have been defined correctly. Consider the pointcut definitions presented in Listing 5.7. If the designer were to accidentally substitute “TaskListneer” instead of “TaskListener” within any of the pointcut definitions, the compiler would not detect the error. However, no checking advice would be bound to the user interface objects, resulting in an incorrect application of the monitor. To overcome this difficulty, designers can use tools such as those provided by the AJDT plug-in for Eclipse [eclipse.org 2004], to statically visualize how advice will be woven throughout a system.
5.6 Chapter Summary

Aspect-oriented programming (AOP) is an approach to improving the modularity of software. AOP techniques are used to factor out concerns that cross-cut class boundaries into compilation units known as aspects. In AspectJ — the de facto standard for AOP-based development — aspects define code fragments called advice that are bound to well-defined execution points by a weaving compiler. The execution points to which advice should be bound are defined by the aspect using pointcut designators.

As design patterns cut across class boundaries, the corresponding proof obligations to be monitored are also cross-cutting. An AOP-based approach that allows for a clear separation of cross-cutting concerns is therefore a natural choice. Using this approach, the contract monitor corresponding to a particular pattern is implemented as an abstract aspect. The advice provided by this aspect performs the monitoring task common across all instances of the pattern in question. The parts that are variable among different applications of the pattern are expressed as abstract methods and pointcuts within the aspect. To tailor the monitor’s behavior to a particular pattern instance, the appropriate definitions are supplied in a subaspect.

The aspect-based monitoring approach offers many of the same advantages as the wrapper-based technique discussed in the previous chapter. However, the approach does not suffer from the same limitations. Namely, the aspect-based approach can be used to monitor the behavior of private methods, and the aspect code has no effect on the identity of objects enrolling or disenrolling from a pattern instance. One important disadvantage of the approach, however, is that it is difficult to determine whether the appropriate pointcuts have been properly defined. This difficulty can be
overcome through the use of specialized development tools that statically visualize how advice will be woven throughout a system.
CHAPTER 6

RELATED WORK

“... in science the successors stand upon the shoulders of their predecessors ...”
— Bertrand Russell.

6.1 Chapter Overview

In this chapter, we briefly survey the literature related to the specification and monitoring techniques presented in this dissertation. The literature summaries are divided among three sections. In Section 6.2, we summarize some of the key literature that our approach to specifying and monitoring design patterns builds on. In Section 6.3, we summarize alternative formalizations of design patterns discussed in the literature, and highlight their principal benefits and limitations. Finally, in Section 6.4, we summarize some of the novel strategies that have been proposed for capturing — in a reusable way — the code required to apply individual design patterns.
6.2 Approach Foundations

6.2.1 Role Models

The work presented in [Reenskaug 1996] introduces the OOram approach to object-oriented analysis, design, and implementation. The essential idea behind the approach is that software systems are composed of objects participating in various areas of concern. For example, in an invoicing system, several objects might collaborate to allow the user to enter a new invoice. Some of these objects might also collaborate to allow the user to print an invoice. Each of these collaborations represents an area of concern in the OOram view. To improve the intellectual manageability of software design — and therefore the resulting quality of the software — the OOram approach handles each area of concern independently. The basic abstraction that enables this isolation — and the cornerstone of the OOram approach — is the role model.

A role model is a projection of a system with respect to a particular area of concern. The model allows the designer to focus only on those aspects of the system relevant to the concern under consideration. Each model consists of a set of roles that characterize the participating objects. The roles do not, however, fully characterize the objects that ultimately implement the design. Instead, a role characterizes only those state components and method behaviors relevant to an object’s participation in a particular collaboration. Or stated another way, a role is a type of filter, through which the designer can focus on those aspects of an object specific to a single area of concern. The OOram method describes notations for documenting role models and the constituent roles, as well as for composing role models to describe more complex patterns of interaction. The notations are generally informal, and bear similarity to those provided as part of the Unified Modeling Language (UML). For example,
the method describes a notation similar to that used in UML class diagrams for identifying the roles that participate in a model, their attributes and methods, and the reference relationships among the roles. Similarly, a notation is provided for characterizing time-ordered sequences of interaction, similar to that used in UML sequence diagrams.

As discussed in Chapter 3, the notion of a role is a key element in our approach to the specification of design patterns. Each pattern implementation may be considered to represent an area of concern. We associate a role with each type of object that participates in an instance of a pattern, and use the role to model those aspects of the object that are material to the pattern’s implementation. We introduce the concept of a *role specification* as a notation for precisely specifying the state components and method behaviors that must be provided by objects playing a particular role. Unlike the informal notations provided by the OOram method, role specifications are formal instruments based on an abstract mathematical view of the objects being specified. Each specification precisely captures the structural and behavioral requirements associated with objects playing a particular role without overly-constraining possible implementations. Additionally, we specify the conditions that must be satisfied for objects to play the role, as well as the conditions that must be satisfied when an object stops playing the role. There are no equivalent notions in the OOram modelling approach. As we have already seen, the role specifications and enrollment conditions are the principal elements of our contract formalism.
6.2.2 Design by Contract

*Design by Contract* (DbC) [Meyer 1987, Meyer 1997] is an approach to improving the correctness of object-oriented software that builds on the pre-condition and post-condition assertions of Hoare Logic [Hoare 1969]. Using DbC, the designer specifies the interface exposed by an object using a *pre-condition* and a *post-condition* associated with each of the object’s methods. These conditions imply a contract that binds the object and its clients. The pre-condition associated with a particular method characterizes the conditions that must be satisfied by the object’s clients before the method can be invoked. Similarly, the post-condition characterizes conditions that are guaranteed by the object when the method terminates — assuming the caller satisfied the appropriate pre-condition. The designer may additionally specify a *class invariant* that governs the contract as a whole. The class invariant specifies conditions that must be satisfied by *every* method exposed by an object.

As discussed in Chapter 3, we make use of the DbC approach in specifying the individual roles that participate in a pattern instance. Each role specification characterizes the methods that must be provided by objects playing the role using standard DbC constructs — pre-conditions and post-conditions. The approach uses these same constructs to specify the conditions that must be respected by player methods used to support areas of concern outside of the pattern’s implementation. Finally, we use a *role invariant* — the role-level analogue of a class invariant — to specify the conditions that must be satisfied by all player methods. The pre-conditions, post-conditions, and invariants used to characterize each role are expressed over the role-level and application-level state of the player objects that ultimately play those roles.
At a higher-level, our specification approach as a whole parallels the structure of DbC. Just as a pre-condition specifies the requirements that must be satisfied by a client when invoking a method, the responsibilities component of a contract specifies the conditions that must be satisfied by a designer when applying a pattern. Similarly, just as a post-condition specifies the guarantee provided by the object in return for the client’s code satisfying the pre-condition, the rewards component of a contract specifies the guarantee provided by the pattern for the designer meeting the specified responsibilities. As discussed in Chapter 3, this similarity is not surprising since patterns are a natural extension of classes.

6.2.3 Parameterized Components

In the RESOLVE [Edwards et al. 1994, Sitaraman and Weide 1994] approach to developing component-based software, components are distinguished as being either abstract or concrete. An abstract component — or concept — is a specification described in terms of a mathematically defined model, and interface operations defined over that model. The fundamental specification approach relies in part on DbC techniques. Component operations are described in terms of pre-conditions and post-conditions, and the operations may be governed by a concept constraint analogous to a class invariant. A concrete component is an executable implementation of its abstract counterpart.

The principal goal of the RESOLVE research effort is to provide a sound foundation for developing reusable software components. As the reusability of the resulting components is one of the key measures of the discipline’s success, abstract components tend to be highly parameterized to prevent specification rigidity. Like most generic
programming efforts, abstract components in RESOLVE are parameterized by constant and type parameters. Additionally, however, components can be parameterized by mathematical definitions, and by other components. Abstract components are expressed over these parameters, definitions of which are supplied as appropriate to a particular component instance. Of course, the designer is not allowed to supply arbitrary actual parameters; the RESOLVE specification language provides constructs to impose constraints on concept parameters.

Similar to abstract components in RESOLVE, design pattern contracts are parameterized by type parameters that are used in specifying the interfaces exposed by the objects participating in a pattern. A more striking — and important — similarity is the parallel between our use of auxiliary concepts and the use of deferred mathematical definitions — also referred to as “math operations” — in RESOLVE. Consider, for example, the RESOLVE sorting component discussed in [Weide, Ogden and Sitaraman 1994]. The authors specify the behavior of the sorting component in terms of the deferred math operation \texttt{ARE.ORDERED}. This operation — which is required to be a total pre-ordering — is used to determine whether two items are in the appropriate sorted order. When defined for a particular component instance, the supplied definition is used to reason about the ordering imposed by the sorter.

As discussed in Chapter 3, we use the same approach in our contract formalism to precisely specify each pattern without compromising the pattern’s flexibility. For example, we use the auxiliary concept Modified — the negation of which is required to be an equivalence relation — to determine what it means for the state of a subject object to be changed in an interesting way.
6.2.4 Checking Wrappers

The work presented in [Edwards et al. 1998] describes a systematic approach to
detecting interface violations in component-based software. The essential idea behind
the approach is based on the use of the Decorator pattern as described in [Gamma
et al. 1995]. Each component is wrapped within another component that exposes the
same interface as the component being wrapped. This parallel interface structure
 guarantees that the wrapped component can be substituted for the original without
modifying the clients that depend on that component. When a call is placed on a
wrapped component, the wrapper checks that the appropriate pre-conditions are sat-
ished before delegating the call to the original. After control returns to the wrapper,
the wrapper checks that the appropriate post-conditions are satisfied before returning
control to the client that initiated the call.

An important feature of the checking approach is that it explicitly distinguishes
between concrete and abstract state. The conditions checked by a wrapper are checked
in terms of the wrapped component’s abstract value. This is achieved by requiring
every checkable component to expose an operation that produces an encoding of its
current abstract value. The checks performed by the monitor are then implemented
over this encoded representation, rather than by invoking methods on the component
itself. Avoiding calls to the wrapped component during the checking process has
two distinct advantages. First, operations exposed by the wrapped component might
themselves be incorrect. While these errors would likely be detected by the wrapper,
the traceability of the detected violations would be limited. Second, operations in-
voked on the wrapped component might cause state changes that are impossible —
or prohibitively expensive — to undo.
In the wrapper-based checking approach discussed in Chapter 4, we leverage the basic checking architecture described above to detect role-level contract violations. In our approach, each checking wrapper is generated dynamically at the point of player enrollment, and is parameterized with respect to the auxiliary concepts used to express the corresponding role-level responsibilities. The assertions checked by the wrapper are based on the concept implementations provided by the designer, and are often unique per player. Like the approach discussed above, we explicitly distinguish between concrete and abstract state, and express the assertion checking behavior over abstract values. To achieve this, we too rely on operations that produce abstract state encodings. Unlike the approach above, however, these operations are not required to be provided by the objects being checked. Instead, these operations are supplied as independent implementations, implemented over the methods of the wrapped player. Of course, this design choice comes at the expense of not benefitting from the advantages discussed above. It does, however, allow us to avoid imposing additional design constraints on the practitioner that might impede the acceptance of the checking approach.

6.3 Design Pattern Formalisms

6.3.1 LePUS


Note that in general, the methods responsible for producing abstract state encodings are not necessarily required to observe the state of the wrapped player through its interface. To avoid making calls to the player during the checking process, the methods can be implemented using a reflection library that allows the internal state of the player to be observed directly. This avoids method calls on the player, but comes at the expense of violating encapsulation — which is likely to have more serious consequences.
used to specify object-oriented architectures and patterns. In LePUS, a design pattern is specified as a formula within the logic. Each formula consists of two parts: (i) a typed list of variable declarations used to capture the participating classes, methods, and inheritance hierarchies, and (ii) a conjunctive statement of the relations among the participants. The ground relations defined by LePUS correspond to standard object-oriented language constructs — and compositions of such constructs. For example, LePUS defines the \textit{Inheritance}(c_1, c_2) relation, used to specify that class $c_1$ must inherit from class $c_2$. Similarly, the \textit{Defined-In}(m, c) relation is used to specify that class $c$ must define method $m$. For convenience, LePUS also includes more complex relations defined in terms of these simple ground relations.

The principal benefit of the LePUS approach is its ability to concisely capture the structural properties relevant to a pattern’s implementation. The approach makes it possible to specify the classes that participate in a pattern, the interfaces they must expose, the relationships among the interface elements, and other structural properties. The approach does not, however, provide adequate support for capturing behavioral properties of interest. Most important, there doesn’t seem to be a mechanism for referring to the state of the participating objects or their call sequences. So, for example, it is possible to specify that a \textit{Subject} method \textit{might} invoke \textit{Notify()}, but the state conditions that prompt the call and the conditions that must be satisfied at the point the call is made cannot be specified. Moreover, the approach neglects the properties that a system will exhibit when a pattern has been applied correctly.
6.3.2 DisCo

*DisCo* [Kurki-Suonio and Järvinen 1989, Järvinen, Kurki-Suonio, Sakkinen and Systä 1990, Järvinen and Kurki-Suonio 1991] is a specification language for characterizing reactive systems that combines an object-oriented view with an action-oriented view. The language is based on an action system model similar to that provided by UNITY [Chandy 1988], and has a formal basis in the Temporal Logic of Actions (TLA) [Lamport 1994]. The essential elements of the formalism are (i) classes, (ii) guarded actions, and (iii) relations. A class declaration describes the data elements provided by objects of a particular type. The declaration does not include any method information, since objects are treated strictly as data elements — they do not provide methods. Instead, individual actions receive objects as parameters, and are responsible for manipulating the data that they contain. A specification may additionally introduce relations that characterize transient associations among groups of objects. Objects can be associated and disassociated with one another through these relations as part of an action’s execution.

The work presented in [Mikkonen 1998] describes the use of DisCo in specifying design patterns. The basic approach is to introduce classes corresponding to the types of objects that participate in each pattern, and actions corresponding to the required methods. Additionally, a specification may introduce relations that capture state and control properties. In the case of the Observer pattern, for example, the specification introduces two such relations: *Attached* and *Updated*. The *Attached* relation is defined over a *subject* and a set of *observers*, and captures those *observers* currently attached to the *subject*. An *observer* is associated with the *subject* through the relation whenever the *Attach()* action is executed, and is disassociated when the *Detach()* action is
executed. Similarly, \texttt{Update} is defined over a \texttt{subject} and a set of \texttt{observers}, and is used to capture the \texttt{observers} that have been \textit{updated} by the \texttt{subject} since the \texttt{Notify()} action was last executed. The relation is cleared for all \texttt{observers} whenever \texttt{Notify()} is executed, and an \texttt{observer} is associated through the relation whenever \texttt{Update()} is executed.

The specification approach succeeds in capturing the temporal properties of interest. It is insufficient, however, as a technique for characterizing the implementation requirements that must be satisfied when applying a particular design pattern, as well as the system properties that are guaranteed by virtue of its application. Most fundamentally, the approach provides inadequate structural guidance. By separating actions from objects — violating a principal tenet of object-oriented design — the resulting specifications do not provide guidance as to how individual classes must be structured. Indeed, a designer might provide an implementation that satisfies the temporal properties characterized by a particular specification, but clearly violates the structural properties that make the pattern a good solution in the eyes of the object-oriented community.

Similar comments apply to the behavioral guidance provided by the formalism. Consider, for example, the case of the Observer pattern. The specification described in [Mikkonen 1998] makes it clear that there is some method — or group of methods — corresponding to the \texttt{Notify()} action. It does not, however, characterize the conditions under which \texttt{Notify()} must be executed, nor does it specify the relevant call sequence conditions that the action must satisfy. Indeed, these conditions are not easily specified using DisCo, since actions cannot invoke other actions directly — action selection is non-deterministic. Moreover, the approach does not consider
methods outside of the pattern’s implementation. Hence, there is no mechanism for imposing conditions on the application-level methods that might interfere with the correct application of a pattern.

Finally, it is worth mentioning that the approach limits the flexibility of design patterns, since DisCo specifications are not parameterized. In the case of the Observer pattern, for example, the specification adopts a definition of consistency that requires the state of every observer to be identical to the state of the subject being observed. This definition of the pattern is of course more restrictive than the original pattern characterization in [Gamma et al. 1995].

6.3.3 BPSL

The work presented in [Taibi and Ngo 2003] is based on the premise that existing pattern specification languages are successful in capturing the relevant structural properties or the relevant behavioral properties — but not both. As a solution, the authors propose BPSL (Balanced Pattern Specification Language) — a specification language intended to capture both the behavioral and structural properties of design patterns. The language combines the key concepts underpinning LePUS and DisCo. To capture the structural aspects of a particular pattern, BPSL uses a subset of First Order Logic, and characterizes the pattern as a formula within the logic. Each formula consists of (i) a list of variable declarations used to capture the participating classes, methods, objects, etc., and (ii) a conjunctive statement of the relations among them. The primary relations defined by BPSL are similar to the ground relations defined by LePUS, and include relations such as Defined-In(m, c) and Inheritance(c₁, c₂) discussed previously. Like LePUS, more complex relations can be defined in terms of these
primary relations. To capture the behavioral aspects of the pattern, BPSL uses an action system model with a formal basis in TLA. While the syntax used to describe the system is different than DisCo, the key concepts are indistinguishable. Indeed, when considering the Observer pattern, the behavioral specification presented in [Taibi and Ngo 2003] is essentially identical to the specification presented in [Mikkonen 1998].

As BPSL derives from LePUS and DisCo, it shares many of the same advantages and disadvantages. With respect to its ability to capture structural properties, BPSL appears to be an effective approach. Moreover, since the approach relies on a subset of First Order Logic, rather than the higher order logic of LePUS, the resulting specifications are generally less complex. Of course, as compared to LePUS, the expressivity of the language is reduced. It is unclear, however, whether the additional expressivity offered by LePUS is required to capture the structural properties of interest. With respect to behavioral properties, the abilities and limitations of BPSL are identical to those of DisCo.

6.3.4 Contracts

The contract formalism discussed in [Helm, Holland and Gangopadhyay 1990, Holland 1992] provides a technique for specifying behavioral compositions in object-oriented software. In this context, a behavioral composition is defined as a group of objects participating to accomplish some task — for example, to implement a particular design pattern. Each contract specifies three key properties: (i) the objects that participate in the composition, (ii) the invariant that the objects maintain.\footnote{Note that the term \textit{invariant} is somewhat misleading here, and is used in this discussion to remain consistent with the work presented in [Helm et al. 1990, Holland 1992]. The authors use the term to refer to a property that the participants try to maintain. The property might, however, become false during the system’s execution.}
and (iii) the conditions that must be satisfied by the participants to instantiate the contract. Each participant is characterized by the attributes and methods that it must provide. Participant methods are specified by a signature, and a sequence of calls and conditions that must be satisfied during the method’s execution. For example, in specifying the Observer pattern\textsuperscript{43} the method used to modify the state of the subject — SetValue() — is required to first update the subject’s value, and then to place a call to Notify(). The contract next specifies the invariant that the participants collaborate to maintain, as well as the sequence of calls required to reestablish the invariant if it is falsified. In the case of the Observer pattern, the contract specifies that every observer will remain consistent with the subject, and that a call to the subject’s SetValue() method will reestablish this invariant. Finally, the contract specifies a sequence of calls and conditions that must be satisfied for the contract to be instantiated. For the Observer pattern, the conditions guarantee that all observers are attached, and that each observer has a reference to the subject.

The contract formalism additionally provides a notation for characterizing how individual classes realize their participant responsibilities in a composition. Each realization is associated with a conformance declaration that describes the mapping between classes and contract participants. A conformance declaration is essentially a collection of bindings between a set of class identifiers and a set of participant identifiers.

The contract approach is similar to our specification formalism, and hence, shares some of the same advantages. There are, however, several important differences. First

\textsuperscript{43}Note that the publication of [Helm et al. 1990] — which presents the specification — precedes the publication of [Gamma et al. 1995] — which presents the pattern. Hence, although the authors describe a pattern that closely resembles Observer, their terminology is not consistent with that used in [Gamma et al. 1995]. We adapt the terminology to remain consistent with previous discussions.
and foremost, the contract language provides inadequate support for specifying conditions on a method’s call sequence. The language allows the specifier to characterize state conditions and method calls that must occur during a method’s execution, as well as the relative order in which these conditions and calls must occur. It does not, however, provide a mechanism for specifying constraints on the conditions and calls that might occur in between or after. Consider, for example, the SetValue() method discussed previously. A realization of this method could satisfy its contract responsibilities by updating the subject’s state, invoking Notify(), and then setting the subject’s state to some random value. The responsibilities would be satisfied since the required state conditions and calls occur in the proper order, but the invariant would clearly be compromised because of the additional action. Similarly, the contract formalism does not provide a construct for imposing conditions on methods outside of the composition\textsuperscript{44}. Consequently, methods provided in support of other compositions can interfere with the composition’s correctness. Finally, while contracts are parameterized by relations analogous to our auxiliary concepts, the parameterization mechanism is underdeveloped. First, relations are associated with the contract as a whole, and cannot be associated with individual participants. More important, the formalism does not allow constraints to be imposed on the allowable definitions. As discussed in Chapter\textsuperscript{3} such constraints are — in general — necessary in guaranteeing that a contract is instantiated correctly.

\textsuperscript{44}It is likely that the absence of such a construct is the primary reason why the authors adopt a definition of invariant that allows the invariant to be falsified, and requires the specifier to identify the sequence of calls required to reestablish the invariant.
6.4 Design Pattern Implementation Strategies

6.4.1 Tricks

In [Eden, Yehudai and Gil 1997, Eden and Yehudai 1997], the authors describe an approach to automating the application of design patterns based on the use of metaprogramming. In their framework, each design pattern is represented as a trick — a metaprogram that operates over the abstract syntax tree of the program in which the pattern should be applied. When a designer wishes to apply a particular pattern, the appropriate trick is selected from a repository, and any required parameters are then supplied. For example, if the trick corresponding to the Observer pattern is selected, the user is required to provide the classes corresponding to the Subject and Observer participant types. This information is used by the trick in determining the appropriate code transformations to be applied.

After a trick has been selected and the appropriate parameters have been supplied, an automation tool is responsible for actually executing the trick. The tool first checks that the supplied parameters satisfy any predefined restrictions imposed on them. Next, the tool produces an abstract syntax tree corresponding to the target system, and executes the trick against this syntax tree. The resulting tree is used to reproduce the system’s code, which now contains an implementation of the selected pattern.

Note that the automation tool can only produce partial implementations for most design patterns since implementations necessarily vary among instances of the same type. To account for these variable aspects, the automation tool may leave portions of the implementation to the designer. In other cases, the tool might provide a default

45The authors of [Eden et al. 1997] note that such restrictions are possible, but do not provide any additional details.
implementation, or might provide a list of alternative implementations from which the
designer can choose. In all of these cases, the designer is free to modify the generated
code as appropriate to the system being developed. The authors do not consider
the question of what responsibilities the designer must satisfy in implementing the
variable portions, nor the question of what effect those implementations might have
on the resulting behavior of the system.

6.4.2 Aspects

Recall that most design pattern implementations cross-cut module boundaries.
That is, the code required to implement a particular pattern is often scattered through-
out multiple classes and/or methods, and is tangled with the code required to support
other areas of concern. To improve the modularity of design pattern implementations,
the work presented in [Hannemann and Kiczales 2002] describes an implementation
strategy based on the use of AspectJ [Kiczales et al. 2001] — an AOP [Kiczales
et al. 1997] language extension for Java46. The authors provide a detailed discussion
of their implementation approach in the context of the Observer pattern, and pro-
vide summaries of their experiences implementing the other 22 patterns characterized
in [Gamma et al. 1995].

Although the details vary, the basic structure of an aspect-based pattern im-
plementation is similar to the structure of an aspect-based contract monitor — as
presented in Chapter 5. The implementation approach essentially factors the state
components and method behaviors typically provided by pattern participants into
an abstract aspect that contains the code common across all instances of the corre-
sponding pattern. Similar to our monitoring approach, the abstract aspect maintains

46The reader is referred to Chapter 5, Section 5.2 for a review of the essential elements of AspectJ.
a map between pattern participants and their corresponding state components. For example, in the case of the Observer pattern, the abstract aspect maintains a map between each subject and its set of attached observers. The aspect provides the methods corresponding to Attach() and Detach() that use this map in providing their functionality. The aspect may also define abstract methods and pointcuts corresponding to those portions of the pattern that vary. For example, in the case of the Observer pattern, the aspect defines an abstract method corresponding to Update(), and an abstract pointcut corresponding to those subject methods that modify the state of the subject. The advice associated with this pointcut is essentially an implementation of the Notify() method, and invokes Update() for each attached observer. Each subaspect implements the abstract methods and pointcuts as appropriate to a particular pattern instance.

While the implementation approach of [Hannemann and Kiczales 2002] bears similarity to our aspect-based monitoring approach, it is worth stressing that their work focuses on how design patterns can be implemented in a modular manner. Our work, on the other hand, focuses on the responsibilities that a designer must satisfy when implementing a particular pattern — in whatever manner — and the rewards to be expected by virtue of its correct application. The aspect-based monitoring approach discussed in Chapter 5 is a testing tool for determining whether the appropriate responsibilities are satisfied at runtime. More simply, an aspect-based monitor is not an aspect-based pattern implementation.

The authors of [Hannemann and Kiczales 2002] note that an alternative approach is to introduce the required state components into the appropriate classes using AspectJ’s static cross-cutting features.
To emphasize this point, it is interesting to note that in evaluating our aspect-based monitoring strategy, we applied a variant of the monitor presented in Chapter 5 to the aspect-based implementation of the Observer pattern presented in [Hannemann and Kiczales 2002]. Surprisingly, when running our test suite, the aspect-based monitor reported a minor violation of the pattern contract presented in Chapter 3. Specifically, the monitor detected that the method corresponding to `Attach()` failed to satisfy the call sequence condition that requires the attaching `observer` to be updated to reflect the state of the `subject` to which it is attaching. Hence, an inconsistent `observer` could attach to the `subject`, and assuming the `subject`'s state was never modified, would remain inconsistent throughout the pattern’s lifetime.
CHAPTER 7

CONCLUSION

“... research is never completed ...”
— Catherine Drinker Bowen.

7.1 Problem Summary

A design pattern describes the core of a solution to a commonly recurring problem in the design of object-oriented software. The benefits of design patterns are two-fold. First, they provide guidance to novice designers on how systems should be constructed. Second, they serve to improve the quality of design documentation. When presented with a new system, knowledge of the patterns applied in its construction should allow a designer to more quickly understand the system’s structure, as well as why it behaves in particular ways.

In the early 1990s, the software engineering community witnessed an increase in research focused on design patterns. Much of this research focused on identifying new design patterns, and presenting those patterns in a standard narrative format. One of the principal results of this work was the publication of numerous pattern catalogs such as [Gamma et al. 1995, Buschmann et al. 1996, Schmidt et al. 2000]. Many of
these catalogs are now considered standard references, and the patterns they contain are an important part of software practice.

Unfortunately, the descriptive format popularized by these catalogs is inherently imprecise. As a consequence, it is not clear when a design pattern has been applied correctly, or what can be concluded about a system implemented using a particular pattern. This ambiguity threatens to undermine the two principal benefits associated with the use of patterns. First, their prescriptive benefits are compromised, as novice designers are not provided with a precise description of how each pattern must be applied, nor the properties that their system will exhibit by virtue of their application. Being novice designers, precise pattern descriptions are especially important, since these designers are most prone to error. Second, the prescriptive benefits of design patterns are compromised, as different designers claiming to have applied the same patterns are likely to mean subtly different things. The resulting design documentation is therefore less effective than it might otherwise be.

7.2 The Thesis

The work presented in this dissertation focused on resolving the ambiguities associated with design pattern descriptions. We described techniques for precisely specifying design patterns, and for determining whether the resulting specifications are respected at runtime. Concretely, the work presented defends the following two-part thesis.

- A parametric specification approach that identifies the structural and behavioral properties of program roles, as well as the interactions among these roles, provides an effective technique for precisely specifying design patterns.
A checking approach based on the use of dynamically generated wrappers or aspect-oriented programming provides an effective technique for monitoring whether design pattern specifications are respected at runtime.

7.3 Summary of Contributions

In support of the first part of the thesis, we presented design pattern contracts in Chapter 3 as a formalism for specifying design patterns. In support of the second part of the thesis, we introduced contract monitors in Chapter 4 as a mechanism for determining whether pattern contracts are respected at runtime. Chapter 4 focused on a monitor design based on the use of dynamically generated checking wrappers [Edwards et al. 1998]. Chapter 5 presented an alternative design based on the use of aspect-oriented-programming (AOP) [Kiczales et al. 1997]. In the subsections that follow we summarize these contributions.

7.3.1 Design Pattern Contracts

A design pattern contract consists of two primary components: a responsibilities component and a rewards component. The responsibilities component of a contract specifies the conditions that must be satisfied when applying a particular design pattern. These conditions include requirements on the structures of participant classes, the behaviors of methods in these classes, and the interactions among the corresponding instances. The rewards component specifies the system properties that are guaranteed to be exhibited if the contract responsibilities are indeed satisfied. In our approach, the reward is captured using a pattern invariant that expresses a property that will remain true whenever the objects participating in the pattern are quiescent.
Note, however, that a pattern contract is not a complete specification of the objects that participate in a pattern. Participating objects typically provide state and behavior to support areas of concern outside of the pattern’s implementation. The state and behavior supplied by these objects in support of additional areas of concern are left unspecified in the pattern contract. To allow for this type of partial specification, we build on Reenskaug’s *role modelling* work [Reenskaug 1996]. In this view, a *role* is a projection of an object with respect to a particular area of concern. We use this concept to model the objects that participate in each pattern. Each participant type is described by a *role-level* specification that specifies only those state components and method behaviors relevant to the pattern’s implementation.

One of the inherent risks in formalizing design patterns is that their hallmark flexibility could be compromised [Riehle 1997]. Each pattern describes a solution that can be customized as appropriate to a wide range of design scenarios. In formalizing patterns, the resulting specifications must not preclude such customization, lest the applicability of the associated patterns be reduced. To achieve precision without sacrificing flexibility, each contract is parameterized by a set of *auxiliary concepts*. Each concept represents a point of variation among instances of the same pattern. The contract is then expressed over these auxiliary concepts, and definitions are supplied per pattern instance. By supplying the appropriate definitions, the designer can tailor each contract on a case-by-case basis.

Of course, the primary advantage of the approach is the precision that it brings to design pattern descriptions. By virtue of being a formal characterization, a pattern contract addresses issues that are not addressed in the informal descriptions. Indeed, the very process of developing a pattern contract forces the designer to resolve
ambiguities that could otherwise lead to an incorrect pattern implementation. For example, recall the Observer pattern contract presented in Chapter 3 and the call sequence requirement imposed on the subject’s Attach() method. This requirement forces (correct) implementations of the Attach() method to invoke Update() on the attaching observer, so as to bring the observer into a state that is consistent with the state of the subject. Although this requirement does not seem to have been discussed elsewhere, it is essential to achieving the pattern’s intent — namely, maintaining consistency between the subject and each attached observer. In the process of developing the Observer pattern contract, the need for this requirement became clear, since the pattern invariant cannot be guaranteed without it.

More surprising than this precision improvement, however, is that contracts actually help to increase the flexibility of individual patterns. This flexibility improvement is achieved through the use of auxiliary concepts that introduce points of variation that are not obviously allowed by the informal pattern descriptions. For example, recall the Memento pattern contract presented in Chapter 3 and the Restored auxiliary concept that captures what it means for the state of the originator to have been suitably restored to a previous value. This point of variation does not seem to be present in the original pattern description. The notion of restoration suggested by [Gamma et al. 1995] requires the state of the originator — or some portion of that state — to be set equal to the value that it held previously. Of course, some might argue that the pattern description presents a canonical case that must be generalized. By making these points of variation explicit, however, designers are encouraged to vary their implementations at these points. The contract serves as a guide, clearly indicating the allowable dimensions of variability.
7.3.2 Contract Monitors

A contract monitor is an executable unit of deployment used to determine whether a contract’s responsibilities are satisfied at runtime. The wrapper-based approach to developing contract monitors relies on programming principles familiar to most designers. The AOP-based approach requires more specialized knowledge, but offers a number of advantages over the wrapper-based approach. Depending on the contract to be monitored, as well as the skill set of the designer, one approach might be preferred over the other. We summarize the key features of each in the subsections that follow.

Wrapper-Based Monitors

When verifying that a pattern implementation correctly realizes its contract, the proof obligations that must be dispatched materialize at three points. In the wrapper-based monitoring approach, the designer is required to place calls to a monitoring library at each of these points. First, at the point of pattern instantiation, the designer must place an instantiation call to the library that results in the creation of a contract monitor corresponding to the newly created pattern instance. Second, whenever an object enrolls in the pattern instance, the designer must place an enrollment call to the corresponding contract monitor. Finally, whenever an object disenrolls from the pattern instance, the designer must place an analogous “disenrollment” call. The appropriate contract responsibilities are checked at each of these points.

Of course, the monitoring library cannot determine at the point of enrollment whether a particular object satisfies its role-level specification. Instead, when an object enrolls to play a particular role in a pattern instance, the enrollment call
results in the generation of a role monitor corresponding to the new player. The purpose of the monitor is to check — throughout the player’s lifetime — whether the player and its clients respect the appropriate role-level specification. The monitor is generated dynamically, and exposes an interface identical to that of the player. The role monitor is thereafter used in place of the player object. When a call is invoked on the player through the role monitor, the monitor intercepts the call, and checks that the appropriate requirements are satisfied. If the monitor detects a contract violation, the violation is reported and the system is terminated. Otherwise, the role monitor is transparent — clients of the player are unaware of the additional levels of indirection.

As design pattern contracts can be customized on a case-by-case basis, the corresponding contract monitors must also be customizable. To allow for this customization, every contract monitor is parameterized with respect to the same auxiliary concepts, abstraction functions, and signature mappings as the contract to which it corresponds. Monitors implement their behavior over abstract methods corresponding to these notions. Just as a designer supplies concrete definitions to customize the contract, they supply concrete method implementations to customize the monitor. By supplying the appropriate implementations, the designer can non-invasively customize the behavior of the monitor as appropriate to a particular pattern instance.

While the wrapper-based approach is appropriate to a large class of systems, there are three important limitations that must be mentioned. First, the approach assumes that after an object enrolls in a pattern instance, the enrolling object will never be accessed without going through the corresponding role monitor. Second, the approach cannot be used to monitor private methods. And finally, when an object enrolls in
a pattern instance, its perceived identity is modified. If the object’s clients depend on this identity, their correctness could be compromised. The AOP-based monitoring approach resolves these limitations.

**Aspect-Based Monitors**

Aspect-oriented programming (AOP) is an approach to achieving a clear separation of concerns in object-oriented software. AOP makes it possible to modularize concerns that are typically considered to be *cross-cutting*. As design patterns describe structures and behaviors that span class boundaries, the proof obligations to be dispatched when showing that a pattern has been applied correctly are also cross-cutting. Since these proof obligations directly correspond to the assertions that must be checked by a monitor, an AOP-based monitoring approach is a natural choice. We focused on developing our monitors using AspectJ — an implementation technology for Java that is the de facto standard for doing AOP-based development.

In the AOP-based approach, a contract monitor consists of a single abstract *aspect* — a compilation unit that modularizes a concern that spans class boundaries. The aspect implements code fragments — called *advice* — that provide the assertion checking behavior appropriate to the corresponding contract. To bind these assertions to the appropriate points in the system’s execution, the aspect additionally defines *pointcuts*. Each pointcut is associated with a piece of advice, and captures a set of execution points at which the advice will be executed.

Of course, the assertions to be checked by a monitor vary not only per pattern, but also per pattern instance. Therefore, following the wrapper-based approach, the checking advice is implemented in terms of abstract methods defined by the aspect. Similarly, the points where assertions should be checked will also vary per pattern
instance. Hence, the pointcuts are also declared as abstract. When a designer wishes to monitor a particular pattern instance, they must provide a subaspect that implements the appropriate methods and pointcuts. The abstract aspect and the subaspect together form the complete monitoring code for one or more pattern instances.

The AOP-based monitoring approach provides the same principal advantages as the wrapper-based approach, but without the associated limitations. Moreover, the development effort required to construct a monitor using this approach is generally less than when using the wrapper-based technique. This comparison holds true both with respect to the quantity of code that must be written, as well as the intellectual complexity of that code. The principal disadvantage of the approach — aside from the requirement that the designer be familiar with AOP principles — is that it is difficult to determine whether the appropriate pointcuts have been properly defined in a subaspect. This difficulty can be overcome through the use of specialized development tools that allow someone to statically visualize how advice will be woven throughout a system.

7.4 Future Work

There are a number of exciting opportunities for further investigation in the areas explored by this dissertation. Some of the areas we plan to explore in the near future are summarized in the subsections that follow.

7.4.1 Contract Families

To capture the flexibility present in the original characterization of a pattern, it may be necessary to introduce several auxiliary concepts in the corresponding contract. Although it is difficult — if not impossible — to measure, the introduction of
each auxiliary concept significantly increases the complexity of the resulting specification. This is problematic given that the primary audience for design patterns is the novice designer. Presented with a specification parameterized by numerous concepts, the designer might be unable to manage the resulting complexity. Of course, the monitoring techniques discussed in Chapters 4 and 5 will help detect contract violations, but it would certainly be preferable that the designer never make such mistakes.

To that end, we plan to explore the notion of a *contract family*. Each family corresponds to a particular pattern, and contains a group of similar contracts that vary in their degree of flexibility — and hence, complexity. A family is produced generatively beginning with the *fully parameterized* contract. New contracts are produced by fixing one or more of the auxiliary concept definitions to capture a common class of pattern instances. For example, in the Memento contract, recall that the auxiliary concept Restored() is used to determine whether the value of the originator at one point is a proper restoration of its value at some previous point. In the common case, restoration requires that the current value of the originator be identical to its previous value. Hence, using equality in place of the Restored() concept — as suggested by the original pattern description — captures a common subset of Memento instances, and reduces the complexity of the contract. We plan to explore this approach in the context of other examples. Our goal is to produce a *contract catalog* that presents a number of families corresponding to some of the most commonly used design patterns. When a designer wishes to apply a particular pattern, they can select the contract

48Note that this approach is similar to the template specialization technique presented in [Bucci, Hollingsworth, Krone and Weide 1994, Weide 2000].
from the corresponding family that best fits the needs of the system and the skills of the designer.

7.4.2 Applicability Issues

In Chapter 3 we used the contract formalism to specify design patterns classified as behavioral patterns in [Gamma et al. 1995]. These patterns are characterized by their descriptive emphasis on the distribution of responsibilities among participants, and the resulting patterns of interaction. Design pattern contracts are well-suited to specifying these patterns, as these are precisely the properties they capture. There are, however, numerous other types of patterns. For example, [Gamma et al. 1995] also identifies structural patterns — focused on how classes and objects should be composed — and creational patterns — focused on how individual objects should be created. Other catalogs present patterns specific to a particular area, such as [Schmidt et al. 2000], which is concerned with patterns for distributed and concurrent computing. Other catalogs focus on patterns related to a particular implementation technology, such as [Alur, Crupi and Malks 2001] and [Thilmany 2003], which present patterns for working with the J2EE enterprise framework and the .NET Framework, respectively. Future work aims to determine whether the contract formalism is appropriate for specifying these other types of patterns. Patterns for concurrent computing are particularly interesting, as these patterns are likely to involve temporal properties that will require additional specification constructs.

7.4.3 Tool Support

In addition to investigating the applicability of the contract formalism to other types of design patterns, we are interested in evaluating the overall utility of the
specification and monitoring approach to software practitioners. Future work aims to determine the productivity and quality improvements achieved when design pattern contracts (and monitors) are used in the construction of actual systems. Like many new development strategies, however, the barriers to adoption pose a difficulty in performing such an evaluation. This is especially true for strategies that possess a formal component, such as ours.

To help overcome adoption barriers, and to thus enable the proposed evaluation studies, we plan to construct a number of development tools that will support our specification and monitoring techniques. First, we plan to develop a library of contract monitors — both wrapper-based and aspect-based — to complement the proposed contract catalog mentioned previously. We plan to integrate these monitors into existing development environments to aid in making pattern testing a standard part of the software development process. Further, we plan to develop tools for automating the construction of these monitors, as well as for automating their application to actual systems.

To automate the monitor generation process, we first plan to precisely characterize the contract language. This language will serve as the basis for a compiler that transforms contract specifications into their monitoring counterparts. Similarly, to automate the application of these monitors, we first plan to precisely characterize the language used to instantiate pattern contracts. This language will provide constructs for describing the classes that participate in a pattern instance, the relevant concept definitions, the relevant abstraction functions, etc. Again, this language will provide the basis for developing a tool that generates the application-specific code required to tailor a contract monitor to a particular pattern instance. The development of this
tool is especially important since applying existing design patterns — as opposed to specifying new design patterns — is the common case.


