

Reverse Engineering of Design Patterns  
from Java Source Code

**Abstract**

Recovering design patterns can enhance existing source code analysis tools by bringing program understanding to the design level. This dissertation presents a new, fully automated pattern detection approach based on our reclassification of the GoF patterns by their pattern intent. We argue that the GoF pattern catalog classifies design patterns in the forward-engineering sense; our reclassification is better suited for reverse engineering. Our approach uses lightweight static program analysis techniques to capture program intent. This dissertation also describes our tool, PINOT, that implements this new approach. PINOT detects all the GoF patterns that have concrete definitions driven by code structure or system behavior. PINOT is faster, more accurate, and targets more patterns than existing pattern detection tools. PINOT has been tested against several benchmark applications, including Apache Ant, Java AWT, JHotDraw, and Swing. Since PINOT has proven successful, we extend PINOT to recognize a broader range of design patterns. This dissertation describes our pattern detection language, MUSCAT, that allows users to define and analyze their own design patterns using the PINOT engine. MUSCAT is a visual language that allows users to model program intent by specifying both the structural- and behavioral- aspects of a design pattern. This dissertation evaluates MUSCAT and discusses the trade-offs between effectiveness and flexibility.

**Reverse Engineering of Design Patterns  
from Java Source Code**

By

NIJA SHI  
M.S. (University of Wyoming) 2001  
B.S. (University of Wyoming) 1999

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Professor Ronald A. Olsson, Chair

---

Professor Premkumar T. Devanbu

---

Professor Zhendong Su

Committee in charge

2007

# Contents

<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background</b>	<b>6</b>
2.1 Critique of Current Approaches . . . . .	6
2.1.1 Targeting Structural Aspects . . . . .	6
2.1.2 Targeting Behavioral Aspects . . . . .	10
2.2 Interpretation and Implementation Variants . . . . .	13
2.3 Motivating Examples . . . . .	15
<b>3 GoF Patterns Reclassified</b>	<b>18</b>
3.1 Language-provided Patterns . . . . .	20
3.2 Structure-driven Patterns . . . . .	21
3.3 Behavior-driven Patterns . . . . .	21
3.4 Domain-specific Patterns . . . . .	22
3.5 Generic Concepts . . . . .	23
<b>4 Approach to Pattern Detection</b>	<b>24</b>
4.1 Detecting Structure-driven Patterns . . . . .	24
4.2 Detecting Behavior-driven Patterns . . . . .	25
<b>5 PINOT</b>	<b>30</b>
5.1 Implementation . . . . .	30
5.2 Results . . . . .	32
5.3 Discussion . . . . .	37
<b>6 Formalizing Design Patterns</b>	<b>39</b>
6.1 Software Modeling . . . . .	40
6.2 Code Refactoring and Generation . . . . .	41

6.3	Pattern Detection . . . . .	43
<b>7</b>	<b>A Comparison of Languages for Pattern Detection</b>	<b>46</b>
7.1	Side-by-side Comparisons . . . . .	46
7.1.1	The Abstract Factory Pattern . . . . .	47
7.1.2	The Observer Pattern . . . . .	50
7.2	Discussion . . . . .	54
<b>8</b>	<b>MUSCAT</b>	<b>58</b>
8.1	Design Guidelines . . . . .	58
8.2	MUSCAT Language Constructs . . . . .	61
8.3	PINOT API . . . . .	61
8.4	Implementation . . . . .	65
8.5	Evaluation . . . . .	67
8.5.1	Defining the GoF Patterns . . . . .	67
8.5.2	Beyond GoF . . . . .	83
8.6	Discussion . . . . .	85
<b>9</b>	<b>Conclusion and Future Work</b>	<b>87</b>
<b>A</b>	<b>Viognier Code Generation – the GoF Patterns</b>	<b>89</b>
<b>B</b>	<b>Viognier Code Generation – Patterns Beyond GoF</b>	<b>106</b>
	<b>Bibliography</b>	<b>110</b>

# List of Figures

1.1	Definitions of false positive rate and false negative rate . . . . .	2
2.1	An Example of a Singleton Class . . . . .	15
2.2	Lazy Instantiation Variant . . . . .	16
2.3	Implementation of <code>getFlyweight()</code> . . . . .	17
3.1	A Reclassification for Reverse Engineering of the 23 GoF Patterns . . . . .	20
4.1	CFG of <code>getTheSpoon()</code> from Figure 2.1 . . . . .	27
4.2	CFG of <code>getFlyweight()</code> from Figure 2.3 . . . . .	28
5.1	Pattern Instances Recovered . . . . .	35
6.1	The Adapter Layer in LayOM [29] Representing the Adapter Pattern . . . . .	42
6.2	FUJABA's Specification for the Abstract Factory Pattern. . . . .	44
6.3	FUJABA's Specification for the Singleton Pattern. . . . .	45
6.4	SPINE's Specification for the Abstract Factory Pattern. . . . .	45
7.1	The OMT Specification for the Abstract Factory Pattern Specification [38] . . . . .	47
7.2	The Abstract Factory pattern in the SPINE Specification [27] . . . . .	48
7.3	The Abstract Factory pattern in the LePUS Specification [13] . . . . .	48
7.4	Partial LePUS Notation Keys [13] . . . . .	49
7.5	The Abstract Factory pattern in the FUJABA Specification [6] . . . . .	49
7.6	The Factory Method pattern in the FUJABA Specification [6] . . . . .	50
7.7	The OMT Specification for the Abstract Factory pattern specification [38] . . . . .	51
7.8	The Observer pattern in the SPINE Specification [27] . . . . .	51
7.9	The Observer pattern in the LePUS Specification [13] . . . . .	52
7.10	FUJABA's specification for the Observer Pattern [6] . . . . .	53
7.11	The Normal Mediator Pattern in FUJABA [6] . . . . .	56
7.12	The OMT Specification of the Mediator Pattern [38] . . . . .	56
8.1	The UML Class Diagram for the Singleton Class [3] . . . . .	60
8.2	Language Constructs for MUSCAT . . . . .	63
8.3	The Implementation of MUSCAT . . . . .	65
8.4	Categorization of the Differing Results . . . . .	69

8.5	The MUSCAT Specification on the Singleton Pattern . . . . .	69
8.6	The MUSCAT Specification on the Facade Pattern . . . . .	69
8.7	The MUSCAT Specification on the Abstract Factory Pattern . . . . .	70
8.8	The MUSCAT Specification on the Visitor Pattern . . . . .	70
8.9	The MUSCAT Specification on the Composite Pattern . . . . .	71
8.10	The MUSCAT Specification on the CoR Pattern . . . . .	72
8.11	The MUSCAT Specification on the CoR Pattern . . . . .	73
8.12	The MUSCAT Specification on the Flyweight Pattern . . . . .	74
8.13	The MUSCAT Specification on the Observer Pattern . . . . .	74
8.14	The MUSCAT Specification on the Mediator Pattern . . . . .	75
8.15	The MUSCAT Specification on the State Pattern . . . . .	76
8.16	The MUSCAT Specification on the Strategy Pattern . . . . .	77
8.17	The MUSCAT Specification on the Bridge Pattern . . . . .	77
8.18	The MUSCAT Specification on the Proxy Pattern . . . . .	78
8.19	The MUSCAT Specification on the Adapter Pattern . . . . .	79
8.20	Timing Results for PINOT vs. cPINOT . . . . .	80
8.21	Timing Results for cPINOT per Pattern (Part I) . . . . .	81
8.22	Timing Results for cPINOT per Pattern (Part II) . . . . .	81
8.23	Timing Results for cPINOT per Pattern (Part III) . . . . .	82
8.24	Variant of Abstract Factory Pattern . . . . .	83
8.25	Variant of Abstract Factory Pattern with Singleton Factory . . . . .	84
8.26	One-to-many CoR . . . . .	84
A.1	The Generated C++ Code that Precedes Every Detection Code . . . . .	90
A.2	The Generated C++ Code that Detects the Abstract Factory Pattern (as defined in Figure 8.7) . . . . .	91
A.3	The Generated C++ Code that Detects the Adapter Pattern (as defined in Figure 8.19) . . . . .	92
A.4	The Generated C++ Code that Detects the Bridge Pattern (as defined in Figure 8.17) . . . . .	93
A.5	The Generated C++ Code that Detects the Composite Pattern (as defined in Figure 8.9) . . . . .	94
A.6	The Generated C++ Code that Detects the CoR Pattern (as defined in Figure 8.10) . . . . .	95
A.7	The Generated C++ Code that Detects the Decorator Pattern (as defined in Figure 8.11) . . . . .	96
A.8	The Generated C++ Code that Detects the Facade Pattern (as defined in Figure 8.6) . . . . .	97
A.9	The Generated C++ Code that Detects the Flyweight Pattern (as defined in Figure 8.12) . . . . .	98
A.10	The Generated C++ Code that Detects the Mediator Pattern (as defined in Figure 8.14) . . . . .	99
A.11	The Generated C++ Code that Detects the Observer Pattern (as defined in Figure 8.13) . . . . .	100

A.12	The Generated C++ Code that Detects the Proxy Pattern (as defined in Figure 8.18)	101
A.13	The Generated C++ Code that Detects the Singleton Pattern (as defined in Figure 8.5)	102
A.14	The Generated C++ Code that Detects the State Pattern (as defined in Figure 8.15)	103
A.15	The Generated C++ Code that Detects the Strategy Pattern (as defined in Figure 8.16)	104
A.16	The Generated C++ Code that Detects the Visitor Pattern (as defined in Figure 8.8)	105
B.1	The Generated C++ Code that Detects a Variant of the Abstract Factory Pattern Pattern (as defined in Figure 8.24)	107
B.2	The Generated C++ Code that Detects a Variant of the Abstract Factory Pattern with Singleton Factory (as defined in Figure 8.25)	108
B.3	The Generated C++ Code that Detects a One-to-many CoR (as defined in Figure 8.26)	109

# List of Tables

2.1	Representative Current Approaches . . . . .	7
5.1	Pattern Recovery Results on AJP . . . . .	33
8.1	Pattern Rules in MUSCAT . . . . .	62
8.2	PINOT API . . . . .	64
8.3	Comparisons on the Abstract Factory Pattern . . . . .	70
8.4	Comparisons on the Visitor Pattern . . . . .	70
8.5	Comparisons on the Composite Pattern . . . . .	71
8.6	Comparisons on the CoR Pattern . . . . .	72
8.7	Comparisons on the Decorator Pattern . . . . .	73
8.8	Comparisons on the Flyweight Pattern . . . . .	73
8.9	Comparisons on the Observer Pattern . . . . .	74
8.10	Comparisons on the Mediator Pattern . . . . .	75
8.11	Comparisons on the State Pattern . . . . .	75
8.12	Comparisons on the Strategy Pattern . . . . .	76
8.13	Comparisons on the Bridge Pattern . . . . .	77
8.14	Comparisons on the Proxy Pattern . . . . .	78
8.15	Comparisons on the Adapter Pattern . . . . .	79
8.16	Results for the Variant of Abstract Factory Pattern . . . . .	84
8.17	Results for the Variant of Abstract Factory Pattern with Singleton Factory . . . . .	84
8.18	Results for the One-to-many CoR Pattern . . . . .	85



## Acknowledgments

The writing of a dissertation can be a lonely and isolating experience, yet my research journey to a PhD degree would not have been possible without the help and support of numerous people. Thus, my sincere gratitude goes to my husband, mother, father, and all my friends for their love and their personal and practical support over the last few years.

My dissertation would not have been possible without the expert guidance and everlasting patience of my advisor, Prof. Ronald A. Olsson. Not only was he readily available for me, as he is for all of his students, but he always read and responded to the countless drafts of my work more quickly than I could have hoped. His comments were always insightful and appropriate. I am very grateful to be able to work with such a brilliant scholar, who taught me how to think like a researcher.

Many thanks go to Prof. Premkumar T. Devanbu and Prof. Zhengdong Su, who served on my dissertation committee. I thank Prof. Devanbu for revising my dissertation under an extremely tight schedule while he was attending a conference in Croatia. I thank Prof. Su. for his suggestions and promptness, despite his very busy schedule, in reading my dissertation. I thank them for their efforts and comments in perfecting my dissertation. I also would like to express my gratitude to Prof. Karl N. Levitt and Prof. Dipak Ghosal, who served on my committee in the qualify exam. I thank them for their insightful suggestions in making the topic broader and applied to other research domains.

Special thanks to Todd Williamson, an undergraduate research assistant who participated in the PINOT project. Without his tireless efforts in verifying correctness of PINOT, the acceptance of our paper submission to ASE'06 would not have been possible.

## Abstract

Recovering design patterns can enhance existing source code analysis tools by bringing program understanding to the design level. This dissertation presents a new, fully automated pattern detection approach based on our reclassification of the GoF patterns by their pattern intent. We argue that the GoF pattern catalog classifies design patterns in the forward-engineering sense; our reclassification is better suited for reverse engineering. Our approach uses lightweight static program analysis techniques to capture program intent. This dissertation also describes our tool, PINOT, that implements this new approach. PINOT detects all the GoF patterns that have concrete definitions driven by code structure or system behavior. PINOT is faster, more accurate, and targets more patterns than existing pattern detection tools. PINOT has been tested against several benchmark applications, including Apache Ant, Java AWT, JHotDraw, and Swing. Since PINOT has proven successful, we extend PINOT to recognize a broader range of design patterns. This dissertation describes our pattern detection language, MUSCAT, that allows users to define and analyze their own design patterns using the PINOT engine. MUSCAT is a visual language that allows users to model program intent by specifying both the structural- and behavioral-aspects of a design pattern. This dissertation evaluates MUSCAT and discusses the trade-offs between effectiveness and flexibility.

---

Professor Ronald A. Olsson  
Dissertation Committee Chair

# Chapter 1

## Introduction

Program understanding tools today are able to extract various source information, such as class structures, inter-class relationships, call graphs, etc. Some may even produce a subset of UML diagrams. However, without proper documentation, it would still take a lot of effort for a developer to become proficient with the source code. Therefore, a powerful program understanding tool should be able to extract the intent and design of the source code. To fulfill this goal, we need some kind of code pattern that bears intent and design as source facts to analyze against. A *design pattern* abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context [38]. A design pattern has its own unique intent and describes the roles, responsibilities, and collaboration of participating classes and instances. Thus, by extracting design patterns from source code, we are then able to reveal the intent and design of a software system. We believe by tracing the common variations of a pattern implementation, the roles of the participating classes can be identified and the intent of the corresponding source code is then revealed.

Every design pattern has its own unique intent. Since design patterns can make software development more efficient and effective, they are widely used in practice. For example, the Singleton pattern is used to implement `java.awt.Toolkit` in the Java AWT package (a GUI toolkit), the Composite, Interpreter, and Visitor patterns form the basic architecture

of Jikes (a Java compiler written in C++) [11], the Flyweight pattern is used in Apache Ant (a Java build tool) to control the helper objects in a project, etc. During the design phase, design patterns serve as a slang for communicating design issues. During the coding phase, design patterns provide clear guidelines on how to create a problem-specific implementation. While design patterns are useful in the forward engineering<sup>1</sup> process, they are equally important in the reverse engineering process. Software projects are usually documented as the software evolves. However, documentation gradually becomes obsolete through time, due to employee turnover or inadequate project management. Further, as the software project grows, legacy code emerges, making software maintenance more difficult without proper understanding of the architectural design. As a result, software companies often find themselves spending lots of time and money on training new developers to get up to speed. Source code contains all the information needed for documentation, but it cannot speak itself without a good reverse engineering tool.

A pattern detection tool can be characterized by its false positive and false negative rates, defined as in Figure 1.1. The false positive rate reflects the degree of soundness, while the false negative rate reflects the degree of completeness of a pattern detection tool. Together the false positive and false negative rates determine the “accuracy” of a pattern detection tool. A good pattern detection tool has low false positive and false negative rates. However, the accuracy rates can vary on different patterns. In general, recognizing program

---

<sup>1</sup>Forward engineering is the process of moving from higher-level abstractions to the actual implementation of a system. An example is code generation.

$$\text{False Positive Rate} = \frac{\text{Number of Incorrect Pattern Instances}}{\text{Number of All Detected Pattern Instances}} \times 100\%$$

$$\text{False Negative Rate} = \frac{\text{Number of Undetected Correct Pattern Instances}}{\text{Number of Correct Pattern Instances}} \times 100\%$$

Figure 1.1: Definitions of false positive rate and false negative rate

behavior is known as an undecidable problem, hence a fully automated static analysis will not be able to achieve 0% false positive and false negative rates.

Past efforts have used structural relationships (such as the generalization and association relationships) to find design patterns. However, pattern detection tools that use structural-based analysis fail to distinguish between patterns that are structurally identical but differ in behavior (e.g., the Strategy and the State patterns). Other approaches attempt to verify pattern behavior using dynamic analysis to distinguish such patterns and to reduce the false positive rate. While these attempts are able to capture program behavior, they fail to interpret and verify program intent that is unique for each design pattern. Furthermore, dynamic analysis depends on the coverage of test data, which can increase the false negative rate if the coverage is not complete.

Design patterns are typically used as guidelines during software development. Thus, the GoF book [38] presents a pattern catalog for forward engineering, but the same classification can be misleading for reverse engineering. Current approaches lack a proper pattern classification for reverse engineering. A pattern classification for reverse engineering should indicate whether or not each pattern is detectable and if there exist traceable concrete pattern definitions to categorize detectable patterns. Thus, we reclassified the GoF patterns into five categories in the reverse-engineering sense (see Chapter 3). Based on this reclassification, we automated the entire pattern recognition process using only static program analysis. This relatively simple approach has proven effective. We have some promising results — both accuracy and speed — from our initial prototype, PINOT (*P*attern *I*Nference and *re*c*O*very *T*ool), in recovering design patterns from the Java AWT package, JHotDraw (a GUI framework [10, 37]), Swing, and Apache Ant. In particular, the Java AWT has been used as a benchmarking suite for pattern detection on Java source code [56, 60]. Our results on speed and accuracy in analyzing the Java AWT are promising.

While PINOT serves its purpose in identifying detectable GoF patterns in Java source code, it is limited by its hard-coded pattern recognition capability. Because a definition

for a design pattern is presented as guidelines, it allows different implementation variant depending on reader interpretation. For example, some studies [26, 27] believe that the Flyweight pattern is a structure- instead of a behavior-oriented pattern. Further, since languages evolve through time, a common implementation for a specific design pattern evolves as well. For example, Java programmers who are acquainted with different versions of the JDK may implement the Iterator pattern<sup>2</sup> differently. Or in many cases, programmers can either choose to use existing implementations of a design pattern provided in a software package or to implement their own. For example, since JDK 1.0, the JDK has provided the implementation of the Observer pattern in `java.util.Observable` and `java.util.Observer`. However, some [7] argue that this implementation is not flexible<sup>3</sup> and prefer to implement the pattern themselves. As a result, different interpretations of a pattern introduce different implementation variants. It is not feasible to hard-code all possible pattern interpretations into PINOT. Moreover, the GoF book only illustrates some of the commonly used design patterns. An active research area in software engineering aims on discovering design patterns used in specific software domains, such as security [19], high-performance computing [53, 16], etc. We want to extend PINOT's pattern recognition capability by allowing users to define and identify their own design patterns. To achieve this goal, we introduce our pattern detection language MUSCAT (*M*inimal *U*ML *S*pecific*C*ATion language). MUSCAT is a visual constraint language that allows users to define the structural and behavioral aspects of a design pattern. Then we define a set of PINOT APIs to facilitate the execution of of the user-defined MUSCAT specifications.

The rest of this dissertation is organized as follows. Chapter 2 critiques current pattern detection tools, discusses pattern interpretation and implementation variants, and presents examples that motivated our approach. Chapter 3 explains our reclassification of the GoF patterns. Chapter 4 illustrates how we identify structure- and behavior-driven patterns.

---

<sup>2</sup>The implementation of the Iterator pattern is provided in `java.util.Enumeration` (since JDK 1.0), `java.util.Iterator` (since JDK 1.2), and as the enhanced for-each loop (since Java 5).

<sup>3</sup>`java.util.Observable` is a class instead of an interface. Thus, Java programmers cannot take an existing class that already extends some other to be an `Observable`, because Java does not allow multiple inheritance.

Chapter 5 describes our initial prototype of PINOT. Chapter 6 discusses various studies on formalizing design patterns. Chapter 7 compares representative languages designed for pattern detection. Chapter 8 presents the design and implementation of the MUSCAT language. Chapter 9 concludes the dissertation and covers our future work.

We reported our initial results on PINOT in Reference [61]. This dissertation contains additional details about our overall approach, a more extensive review of background, more examples, further information on the PINOT implementation, further experimental results, and additional discussion. It also presents our work beyond PINOT, which defines our pattern detection language MUSCAT.

# Chapter 2

## Background

### 2.1 Critique of Current Approaches

Approaches to design pattern recognition fall into two main categories: those that identify the structural aspect of patterns and others that take a further step to distinguish the behavioral aspect of patterns. Table 2.1 summarizes representative current approaches. For each pattern detection tool, the target language, detection techniques, case studies (generally some well-known applications), and the patterns identified in the case studies are shown.

#### 2.1.1 Targeting Structural Aspects

These approaches analyze inter-class relationships to identify the structural aspect of patterns, regardless of their behavioral aspect. The targeted inter-class relationships include: class inheritance; interface hierarchies; modifiers of classes and methods; types and accessibility of attributes; method delegations, parameters and return types.

Previous work [26, 23, 48, 56, 25, 60, 40, 57, 65, 62, 59, 28] uses structural analysis to find GoF design patterns [38] from source. Structural relationships of the code include class inheritance, interface hierarchies, attributes, method invocations, parameters and return



Tools	Language	Techniques	Case Studies	Patterns Identified
SPOOL [48]	C++	Database query	ET++, two classified systems from Bell Canada	Template Method, Factory Method, Bridge
DP++ [26]	C++	Database query	DTK	Composite, Flyweight, Class Adapter
Vokac et al. [65]	C++	Database query	SuperOffice CRM	Singleton, Template Method, Decorator, Observer
Antoniol et al. [23]	C++	Software metric	Leda, libg++, socket, galib, groff, mec	Bridge, Adapter
SPQR [62]	C++	Formal semantic	Some C++ test programs	Decorator
Balanyi et al. [25]	C++	XML matching	Jikes, Leda, Star Office Calc, Writer	Builder, Bridge, Prototype, Proxy, Strategy, Template Method, Factory Method
PTIDEJ [22, 40]	Java	Constraint solver	Java AWT, java.net package	Composite, Facade
FUJABA [56, 57, 67]	Java	Fuzzy logic and Dynamic analysis	Java AWT	Bridge, Strategy, Composite
Heuzeroth et al. [46]	Java	Dynamic analysis	Java Swing	Observer, Mediator, CoR, Visitor
HEDGEHOG [28, 27]	Java	Formal semantic	AJP [63] code sample, PatternBox [17], Java Language 1.1 and 1.2	Abstract Factory, Factory Method, Prototype, Singleton, Adapter, Bridge, Composite, Decorator, Flyweight, Proxy, Iterator, Observer, State, Strategy, Template Method, Visitor
KT [30]	SmallTalk	Dynamic analysis	KT, three SmallTalk programs	Composite, Decorator, Template Method
MAISA [58]	UML	UML matching	Nokia DX200 Switching System	Abstract Factory
Tsantalis et al. [64]	Java	Graph matching w/ similarity scoring	JHotDraw, JRefactory, JUnit	Adapter/Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, Strategy/State, Template Method, Visitor

Table 2.1: Representative Current Approaches

types, object creations, and variable access within a method.

Some approaches first extract inter-class relationships from source code and then perform pattern recognition based on the extracted information. For example, DP++ [26], SPOOL [48], Osprey [24], and Reference [65] extract inter-class relationships from C++ source to a database; patterns are then recovered through queries to the database. Reference [36] combines the Columbus reverse-engineering framework with the MAISA architectural metrics analyzer (which analyzes software at the design level and had reported limited results on recovering anti-patterns [58]) to build a pattern recognizer. However, pattern recognition requires analyzing program behavior, which can be abstracted away at the design level.

Reference [25] uses the Columbus schema for the extracted abstract semantics graphs<sup>1</sup> (ASG) and recovers patterns based on graph comparison. Reference [23] extracts inter-class relationships and then uses software metrics to reduce search space. Reference [64] analyzes Java bytecode using graph matching based on similarity scoring. In contrast to an exact graph matching, their approach uses inexact graph matching algorithms to facilitate detection of implementation variants. However, their approach does not consider the behavioral aspect of patterns, thus it is limited to detecting patterns based on their inter-class relationships. While Table 2.1 indicates that their approach detects what we later categorize as *behavior-driven* patterns (in Section 3.3), their search criteria does not involve any semantic constraints. Such discrepancies are quite common among pattern recognition tools. Section 2.2 discusses pattern interpretation in more detail.

SOUL [34] is a logic inference system, which has been used to recognize patterns (in Java and SmallTalk) based on inter-class-based code idioms and naming conventions. SPQR [62] uses denotational semantics to find patterns on the ASG obtained by gcc. The accuracy of these approaches depends in part on the capability of the program facts extractors they use. Although extracting inter-class relationships seems straightforward, it is

---

<sup>1</sup>The abstract semantic graph here is an enriched abstract syntax tree generated by Columbus and later processed by MAISA.

complicated by variations in the implementations of some relationships, such as aggregation [60, 56]. Thus, these approaches can result in higher false positive or false negatives rates.

FUJABA [56] extends the work from [60] and uses a bottom-up-top-down approach to speed up the search and to reduce the false positive rate (due to more complicated inter-class relationship, such as aggregation [60, 56]). It uses a combination of inter-class relationships to indicate a pattern. Thus, when such information is obtained from the bottom-up search, even partially, FUJABA assumes the existence of a possible pattern and tries to complete the rest of the search — i.e., the top-down search — to confirm that such a pattern actually exists. This iterative approach allows going back to their annotated abstract syntax tree (AST) for further analysis on demand.

From a system’s architectural description (which includes UML class, activity, component, and sequence diagrams), MAISA is able to find design patterns and anti-patterns [31]. However, they targeted only the Abstract Factory pattern. Only the Abstract Factory pattern (which represents a “good” pattern) and the Blob anti-pattern<sup>2</sup> (which represents a “bad” pattern) were found in their analyzed system.

Recovering design patterns from architectural descriptions is ineffective in practice for two reasons. First, during software development, architectural requirements and descriptions are usually laid out at the beginning of the development cycle, but are rarely reiterated and detailed as the project evolves. Second, to use MAISA to find patterns, one needs to first extract a set of UML diagrams (including both structural and behavioral diagrams) from source; however, how to recover system behavior is still ongoing research in the UML community.

---

<sup>2</sup>The Blob pattern describes the lack of OO design, which requires refactoring techniques to break the blob into object components. However, this work identifies the “Blob” when unsynchronized shared memory is found using the UML component and sequence diagrams.

### 2.1.2 Targeting Behavioral Aspects

The approaches discussed in Section 2.1.1 are unable to identify patterns that are structurally identical but differ in behavior, such as State vs. Strategy and Chain of Responsibility (CoR) vs. Decorator. Approaches that target behavioral aspects seek to resolve this problem using machine learning, dynamic analysis, and static program analysis.

#### Machine Learning

These approaches attempt to reduce false positives by training a pattern recognition tool to identify the correct implementation variants of a pattern. Such approaches are semi-automatic: user intervention guides pattern recognition.

Like Reference [48], FUJABA is a semi-automatic detection tool. They believe pattern recognition is driven by a semi-automatic iterative process. Follow-on work of FUJABA [57] associates fuzzy values to pattern definitions. During the recognition process, the fuzzy values may be updated at each iteration. The pattern detection engine is bundled with the FUJABA Tool Suite RE (a software round-trip engineering tool for Java), which is in parallel with the work in Reference [22]. The pattern recognition process in FUJABA's more recent work [55] suggests a more user-driven approach. They believe pattern detection requires human intervention to overcome scalability problems caused by implementation variations in different problem domains. Thus, this approach assumes users to have a fair amount of knowledge of the analyzed code. However, reverse engineering tools for design patterns are typically used in understanding legacy code, where users may not be able to provide any feedback during the reverse engineering process.

PTIDEJ [22] recognizes distorted implementations of patterns, thus detected pattern instances are associated with a similarity rate. A related work of PTIDEJ [41] uses program metrics (such as size, cohesion, and coupling) and a machine learning algorithm to *fingerprint* roles of a pattern's participating classes. These fingerprints are learnable facts for the pattern constraint solver.

Reference [35] (follow-on to [25, 36]) incorporates machine learning techniques to train its pattern recognition tool. Each pattern is defined with a set of predictors, whose values are used in the learning process. They tested their method on the Adapter and Strategy patterns.

Most GoF patterns (including the Adapter and Strategy patterns) have concrete definitions on their realization in code structure and system behavior. Such concrete definitions are traceable (see Chapter 3). Thus this category does not seem to solve the fundamental problem (see further Section 5.2).

### **Dynamic Analysis**

Some design patterns include specification of program behavior, which cannot be determined analyzing only structural relationships. These approaches use runtime data to help identify the behavioral aspects of patterns.

KT [30] hard-coded its detection algorithms to search for patterns in programs written in SmallTalk. It avoids detecting patterns that are structurally identical, e.g., the Strategy, State, and Command patterns. These patterns share a common idea — the reification of responsibility. However, these patterns have very different intents and, therefore, very different behaviors. KT uses only dynamic analysis to identify the CoR pattern, but the result was unsuccessful, KT failed to find the Chain of Responsibility (CoR) pattern. KT's search algorithm for the pattern is based on only dynamic analysis. It analyzes an object-message diagram interpreted from a call tree constructed during runtime. This process removes unnecessary message calls unrelated to the search. Then the pattern should be identified if the object-message diagram captures the right pattern behavior. However, this approach failed to find the Chain of Responsibility pattern due to improper message logging mechanism and insufficient test data.

Follow-on work to FUJABA [66] and Reference [46] suggest using dynamic analysis to analyze behavior. First, they obtain inter-class information from source code. Next, for

a particular pattern, they compute a list of candidate classes. Then, assuming how these candidates should behave, they verify the behavior during runtime. In particular, Reference [66] uses dynamic analysis as part of pattern identification and incorporates UML sequence diagrams to specify behavioral aspects. This approach complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. We believe that structural analysis should be used to narrow down the search space. Without any experimental results or proof, these approaches References [46] and [67] claim that traditional data-flow and control flow analysis should not be feasible when polymorphism and dynamic method binding are involved. However, the critical behavior in a design pattern is defined in the base class. Therefore, we rarely have to trace every possible path happening in the subclasses(s) (see the Chain of Responsibility pattern and other behavioral patterns in [38]).

Dynamic analysis relies on a good coverage of test data to exercise every possible execution path; such test data is not often available. Even if test data is available in a distribution, the runtime results may be misleading since the data was not originally designed for recognizing the behavior of a particular pattern (e.g., a distribution might include a validation or benchmark suite). Moreover, dynamic analysis is not able to verify pattern intent that is not observable, such as verifying *lazy instantiation* and *single instance assurance* for the Singleton pattern,

### **Static Program Analysis**

These approaches apply static program analysis techniques to the AST in method bodies. FUJABA, in its current implementation, identifies path-insensitive object creation statements for recognizing Abstract Factory and Factory Method patterns. Reference [28] is a design pattern verification tool for Java. The tool consists of the HEDGEHOG proof engine and the Prolog-like SPINE specification language. HEDGEHOG verifies whether a Java class definition correctly implements a particular design pattern defined in SPINE.

In particular, HEDGEHOG identifies inter-class relationships and then applies some inter-procedural but path-insensitive analysis techniques to verify some weak semantics (e.g., whether a method modifies the value of a field) defined in method bodies. SPINE is not able to capture program intent, thus patterns that are vaguely defined or lack clear realization are not representable in SPINE. Design patterns shown in Table 2.1 for HEDGEHOG are the representable ones that have been successfully verified. HEDGEHOG has an accuracy rate of 85.5% for all SPINE-representable patterns. Since the analysis for verifying weak semantics are hard-coded in HEDGEHOG, the false negatives comes from HEDGEHOG's limitation of recognizing implementation variants (see further Section 5.2).

## 2.2 Interpretation and Implementation Variants

During our research in reverse engineering design patterns from source code, we have encountered various forms of reification of the same design pattern. Each design pattern listed the GoF book [38] has an *intent*, a short description explaining the pattern's purpose and implementation. Based on the intent, the GoF book then describes the implementation guidelines and possible implementation variants for programmers to adopt a particular design pattern. Programmers apply design patterns at various stages of software development. And very often, design patterns come into play in the course of refactoring [49]. Because the GoF book only offers guidelines, instead of a precise definition, it gives programmers the confidence and flexibility to improve their code. However, as a vendor who builds pattern recognition tools, we need a precise definition for each pattern specifying the exact criteria that distinguishes its implementation from other code. Reference [42] (the author of PTIDEJ) addresses the problem of interpretation of the UML specifications for the design patterns presented in the GoF book. They argue that there is a gap between modeling and programming on the binary relationships (i.e., UML associations) in the UML class diagrams. This gap introduces a wide range of implementation variants, thus the UML

binary relationships need to be formalized.

However, the vagueness of the unformalized UML binary relationships is not the main issue that opens the door to a variety of pattern implementations. It is mainly due to the intentionally vague guidelines presented in the GoF book. As a result, various interpretations of a design pattern emerge and each interpretation leads to an implementation variant. An example of an implementation variant due to interpretation is the Flyweight pattern. The GoF book describes its intent as “[to] use sharing to support large numbers of fine-grained objects efficiently”. References [47, 27] consider immutable classes (e.g., `java.lang.String`) as an implementation of the Flyweight pattern; Reference [26] believes the Flyweight pattern shares the same structural implementation with the Composite pattern. The sample code illustrated in the GoF book suggests an implementation that includes a flyweight factory that creates and manages a pool of flyweight objects. However, based on our findings (see Section 5.2 for details), the GoF version of the implementation turns out to be hardly used in practice.

Ideally, a design pattern should only have one interpretation and allow different implementation variants. Then, the effectiveness of a pattern recognition tool is determined by how broad a tool is capable of recognizing possible implementation variants. For instance, to implement the GoF version of the Flyweight pattern, it is the programmer’s choice to use any data structure (e.g., a vector, hash table, etc.) to implement the one-to-many relationship (such UML binary relationship is also discussed in Reference [42]) for the flyweight object pool. Such implementation variants derive from the same interpretation and should be recognized by a pattern recognition tool that defines search criteria based on this interpretation.

Unfortunately, a design pattern is defined by an intent and different interpretations and implementation variants are unavoidable. Chapter 6 discusses various studies on formalizing design patterns.



## 2.3 Motivating Examples

Current pattern recognition approaches fail to properly verify pattern intent, which is an important aspect of patterns.. For example, the Singleton and the Flyweight patterns are both object-creational patterns, but each has a unique intent that can be implemented in various ways.

### Example: the Singleton Pattern

The Singleton pattern is probably the most commonly used pattern. Figure 2.1 (based on [5]) shows a common implementation of the Singleton pattern. It is generally perceived to be the simplest pattern to detect [65, 59], since it does not require analyzing its interaction with other classes. The intent of the Singleton pattern is to ensure that a class has only one instance [38]. However, to verify this intent is not an easy task and is typically omitted or limited in current recognition tools.

```
public class SingleSpoon {
    private SingleSpoon();
    private static SingleSpoon theSpoon;
    public static SingleSpoon getTheSpoon() {
        if (theSpoon == null) theSpoon = new SingleSpoon();
        return theSpoon;
    }
}
```

Code based on <http://www.fluffycat.com/Java-Design-Patterns/Singleton>

Figure 2.1: An Example of a Singleton Class

For example, FUJABA's recognition is solely based on inter-class relationships, which identifies a Singleton class with the following criteria: (1) has class constructors regardless of accessibility, (2) has a static reference, regardless of accessibility, to the Singleton class, and (3) has a public-static method that returns the Singleton class type. Thus, without further static behavioral analysis in the method bodies, FUJABA identifies a Singleton class as long as it matches these constraints. In fact, (1) and (2) are incorrect. The constructors have to be declared private (unless the class is abstract) to control the number of objects created, and the Singleton reference also has to be private to be prevent external modification. Even

with these constraints modified, the Singleton class structure only prevents external instantiation and modification. The real pattern intent is embedded in the public-static method's body.

As another example, HEDGEHOG uses limited static semantic analysis to verify implementation of lazy instantiation (illustrated in `getTheSpoon()` of Figure 2.1). The lazy-instantiation analysis is hard-wired in HEDGEHOG. Based on the other semantic analysis techniques discussed in Reference [28], HEDGEHOG is not able to recognize other forms of lazy instantiation, such as using boolean (or other data types) flags to guard the lazy instantiation or using a different program structure (illustrated in Figure 2.2, based on [5]).

```
public static SingleSpoon getTheSpoon() {  
    if (theSpoon != null) return theSpoon;  
    theSpoon = new SingleSpoon();  
    return theSpoon;  
}
```

Figure 2.2: Lazy Instantiation Variant

### Example: the Flyweight Pattern

The Flyweight pattern has various interpretations. Because it is categorized as a structural pattern by GoF, many believe the pattern is based on inter-class relationships. However, the pattern consists of a flyweight factory (that manages a pool of sharable-unique flyweight objects), which makes it more of a creational pattern and thus requires verifying pattern intent. The GoF book specifies that a flyweight object is created upon request. Each created flyweight object stored in a flyweight pool is associated with a unique key for later retrieval. Figure 2.3 (based on [8]) shows an implementation of the GoF interpretation. The real pattern intent resides in the method body of `getFlyweight(...)`.

FUJABA interprets a flyweight class of having a container and a method that keeps and creates flyweight objects, respectively. This strategy fails to capture the pattern intent and significantly increases the false positive rate.

```
public class FlyweightFactory {
    Hashtable hash = new Hashtable();
    public BallFlyweight getFlyweight(
        int r, Color col, Container c, AStrategy a) {
        BallFlyweight tempFlyweight =
            new BallFlyweight(r, col, c, a),
        hashFlyweight =
            ((BallFlyweight)hash.get(tempFlyweight));
        if(hashFlyweight != null) return hashFlyweight;
        else {
            hash.put(tempFlyweight, tempFlyweight);
            return tempFlyweight;
        } } }
```

Code based on

<http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/FlyweightPattern.htm>

Figure 2.3: Implementation of `getFlyweight()`

HEDGEHOG, on the other hand, interprets an immutable class as a flyweight class and any `static-final` variables as sharable flyweight objects. However, such interpretations tend to be overly restrictive and fail to recognize the GoF interpretation of the Flyweight pattern.

## Chapter 3

# GoF Patterns Reclassified

The GoF book [38] illustrates 23 common design patterns and categorizes them based on their purposes and scopes. Purposes have three categories: creational, structural, and behavioral. Creational patterns focus on how objects get created. Structural patterns focus on class organization by roles using structural relationships, such as class inheritances, interface hierarchies, and attribute associations. Behavioral patterns focus on separating object responsibilities based on polymorphism and delegation. Patterns are grouped by their scopes into either object or class patterns. Class patterns deal with relationships between classes. This relationship is established statically during compile-time through inheritance. Object patterns deal with dynamic relationship between objects during runtime. Most patterns described in the GoF book are object patterns. Based on the GoF categorization, some researchers [25, 66] believe structural patterns can be identified based on only inter-class relationships and require the least effort to analyze. Creational patterns come next, since statements of object creation can be easily detected. Behavioral patterns are considered the most difficult to detect, since analysis on the behavior in the method body is required. However, that view is not entirely accurate. As discussed in Section 2.3: the Singleton pattern (a creational pattern) requires not only detecting the existence of object creation, but it also requires verifying the behavior of the method body that creates and returns the Sin-

gleton instance; the Flyweight pattern (a structural pattern) requires behavioral analysis to verify whether all flyweight objects in the flyweight pool are singletons and are created on demand. The Template Method and Visitor pattern (both behavioral patterns) define their behavior in the class definitions, which can be identified based on static structural analysis (see Sections 3.2). While this categorization is useful for programmers, it is not helpful for pattern detection.

Instead of using purposes and scopes, patterns should be categorized, in the reverse-engineering sense, by their definitions from the structural and behavioral aspects. Some patterns are driven by code structure and are designed to structurally decouple classes and objects; but, other patterns are driven by system behavior and require specific actions implemented in the method bodies. As mentioned in Section 2.1, a reclassification of design patterns in the reverse-engineering sense is needed. Thus, we divide the GoF patterns based on their structural and behavioral resemblances into five categories: patterns that are already provided in the language (Section 3.1); patterns that are driven by structural design and can be detected using static structural analysis (Section 3.2); patterns that are driven by behavioral design and can be detected using static behavioral analysis (Section 3.3); patterns that are domain-specific (Section 3.4); patterns that are only generic concepts (Section 3.5). Figure 3.1 illustrates this reclassification and highlights our search strategies. The squares are the design patterns, and ovals are structural sub-patterns (which are the building blocks of the design patterns; some of the sub-patterns here are used in References [56, 62]). The un-boxed texts indicate the searching criteria along the edge to another design pattern. A standalone square indicates that the pattern is detectable either by its inter-class properties or using additional domain-specific knowledge and heuristics.

The following sections are organized based on this categorization. Each section discusses the common characteristics and search strategies of the patterns in a category.

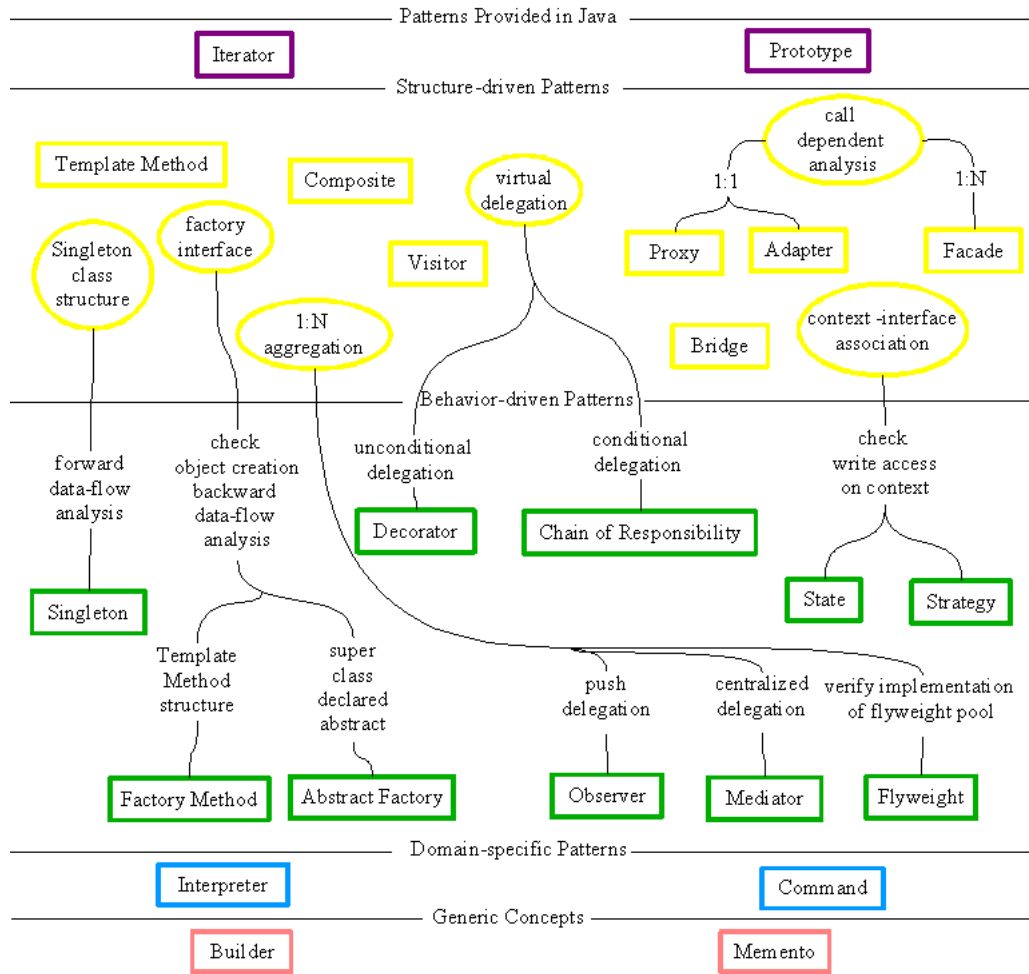


Figure 3.1: A Reclassification for Reverse Engineering of the 23 GoF Patterns

### 3.1 Language-provided Patterns

Our idea of reverse engineering design patterns from source code is not limited to detecting patterns written in a particular programming language (although so far we have focused on Java). Design patterns are so widely used today that many languages (e.g., Java, Python) and packages (e.g, JDK, STL) implement some common design patterns to facilitate programming. Java provides the Iterator (as in `java.util.Enumeration`, `java.util.Iterator`, and the `for-each` loop) and Prototype (as the `clone()` method in `java.lang.Object`) patterns. In practice, developers tend to use such built-in facilities<sup>1</sup> to efficiently and effectively

<sup>1</sup>`java.util.Observable` implements the Observer pattern; it implements a fixed subject-listeners communication mechanism, and the order in which notifications will be delivered is unspecified (see the Java2 API). In

build software systems. Such pattern instances can be recognized by matching specific names for methods or checking if a class implements a specific Java interface, which is used in HEDGEHOG [27]. The latter is used in HEDGEHOG [28]. For example, HEDGEHOG identifies an Iterator class by verifying whether it *implements* `java.util.Iterator` or `java.util.Enumeration` and identifies a Prototype class by checking if it *implements* `java.util.Cloneable`.

## 3.2 Structure-driven Patterns

Patterns in this category can be identified by inter-class relationships. Such relationships establish the overall system architecture but do not specify the actual system behavior. Inter-class relationships are used to separate class responsibilities that contain declarations, generalization, association, and delegation relationships. Structure-driven patterns include the Bridge, Composite, Adapter, Facade, Proxy, Template Method, and Visitor patterns. The Bridge and Composite patterns separate class hierarchies based on generalization and association relationships; the Adapter, Facade, and Proxy patterns separate class roles based on class association and method delegation relationships; the Visitor and Template Method patterns defer class responsibilities through method declarations and delegation.

## 3.3 Behavior-driven Patterns

Some patterns are designed to realize certain behavioral requirements. Such a design pattern is embedded with a program intent that is carried in inter-class relationships and method bodies. Behavior-driven patterns include the Singleton, Abstract Factory, Factory Method, Flyweight, Chain of Responsibility (CoR), Decorator, Strategy, State, Observer, and Mediator patterns.

---

practice, the Observer pattern is applied to various contexts with different internal data structures and communication mechanisms. Thus, we include the Observer pattern in the Behavior-driven Patterns category (Section 3.3).

The GoF creational patterns are driven by some constraints on object creation, such as the number and type of objects to be created. For example, the Singleton pattern ensures that a class has at most one instance during the entire program execution. The Flyweight pattern, although classified as a GoF structural pattern, is designed to effectively manage a pool of sharable objects.

The CoR and Decorator patterns define different behavior based on how a request is passed along a list of handlers. The Decorator pattern lets every handler process the same request, while the CoR pattern passes a request along the list until the right handler processes it.

The Strategy and State patterns share identical inter-class structures but differ in behavior. Each pattern involves a context class that has an attribute that takes a role of either a strategy or a state. The two patterns differ in how the attribute gets modified. In the State pattern, the attribute is passively modified by other state objects. In the Strategy pattern, the attribute is actively modified by other class entity through the context class.

The Observer (subject vs. observers) and Mediator (mediator vs. colleagues) patterns share the same 1:N aggregation relationship but differ in communication styles. The subject class of the Observer pattern broadcasts messages to its observers, while the mediator of the Mediator pattern serves as a communication hub for its colleagues.

### **3.4 Domain-specific Patterns**

The Interpreter and Command patterns combine other GoF patterns and are specialized to suit a particular domain. The Interpreter pattern uses the structure of the Composite pattern and the behavior of the Visitor pattern. Based on this formation and a grammar of a language, the Interpreter pattern interprets the language. We consider this pattern as a special case of realizing the Composite and the Visitor patterns. The Command pattern is basically a realization of the Bridge pattern that separates the user interface from the actual



implementation for command execution. The Command pattern also suggests incorporating the Composite pattern to support multi-commands and undoable operations and using the Memento pattern to store the history of executed commands. Such patterns are possible to detect, but their detection requires analysis that incorporates domain-specific knowledge. Thus, no reverse-engineering work has targeted them.

### 3.5 Generic Concepts

While useful in practice, the Builder and Memento patterns are only generic concepts lacking traceable implementation patterns. The Builder pattern is a creational pattern that separates the building logic from the actual object creation, so that the building logic is reusable [38]. In practice, this pattern is often used for system bootstrapping, of which object creation may not be involved with initial configuration. The Builder pattern was detected in Reference [25] (as shown in Table 2.1) with a 86% false positive rate. The Memento pattern “captures and externalizes an object’s internal state so that the object can be restored to this state later” [38]. However, the pattern neither defines the representation for a state nor the requirement of a data structure for the memo pool. This pattern has not been addressed in any pattern detection tools discussed in Section 2.1, because similar to the Builder pattern, these patterns are generic concepts that lack definite structural and behavioral aspects for pattern detection.

## Chapter 4

# Approach to Pattern Detection

A design pattern is an abstraction of source code design and can be realized in many ways, which makes it non-trivial to detect. However, a pattern can be effectively detected using various program analysis techniques if it has a concrete definition of how it realizes its structural and behavioral aspects. Thus in our current scope, we exclude detection for domain-specific patterns (which requires the specific domain-specific knowledge to be further defined) and generic concepts (which lack clear structural and behavioral definitions). We also exclude language-provided patterns, since they are included in the language and require only trivial keyword analysis. In this dissertation, we focus on detecting the structure- and behavior-driven patterns.

### 4.1 Detecting Structure-driven Patterns

Section 3.2 discussed how such patterns can be detected by their inter-class relationships. Information on various inter-class relationships can be obtained through parsing. Then, specific analysis is applied to different patterns.

The Bridge, Composite, and Template Method patterns have been successfully identified in previous work (that target structural aspects) based on inter-class relationships. We use the same approach in this case.

The Visitor pattern provides a way to define a new operation to be performed on an already-built object structure without changing the classes of the elements on which it operates [38]. The inter-class relationships involved are: a method declaration `accept` (e.g., `void Accept(Visitor v)`), defined in the element class to invite a visitor; and a method invocation `visit` (e.g., `v.visit(this)`), where an element exposes itself to the visitor.

The Object Adapter (adapter vs. adaptee), Facade (facade vs. subparts), and Proxy (proxy vs. real) patterns share a common goal: to define a new class to hide other class(es) for system integration or simplification. We will refer to the Object Adapter pattern as the Adapter pattern. The Adapter and Proxy patterns each hides one class, whereas the Facade pattern hides multiple classes (to be distinguished from the Adapter pattern). By “hiding”, we mean the hidden classes are not directly accessed (by reference or delegation) from others except for the one that is hiding.

Some other basic inter-class structures also need to be identified for detecting behavior-driven patterns (see further Section 4.2). The Singleton class structure is based on the structural features described in Section 2.3. The sub-patterns (as the ovals in Figure 3.1) are also identified for further behavioral analysis. These inter-class sub-patterns can be identified by analyzing class inheritance, class and method declarations, and method delegations. The next section further discusses these structures.

## 4.2 Detecting Behavior-driven Patterns

The inter-class analysis (defined in Section 4.1) identifies the structural aspect of a pattern, and most importantly narrows down our search space to particular methods for further static behavioral analysis. For example, identifying the Singleton class structure determines whether the singleton instance is created (1) once upon declaration or (2) by lazy instantiation. Then, static behavioral analysis is applied to each candidate method’s body to verify whether for (1) it simply returns the instance or for (2) it correctly implements

lazy instantiation.

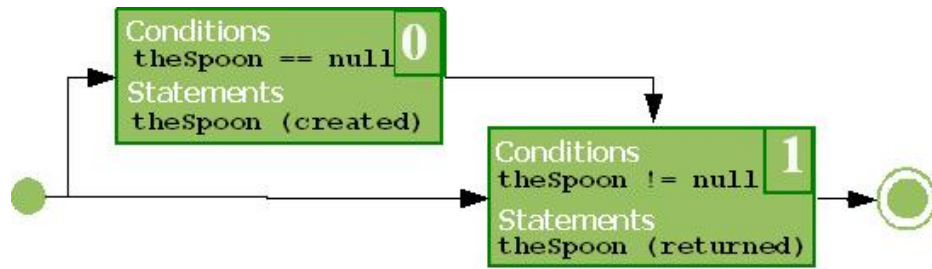
There are several ways to understand program behavior. A common technique is template matching, which is often used in detecting malicious or buggy code (e.g., [44]). If applied to pattern detection, we can perhaps characterize certain pattern behavior into a sequence of states, then make it a template to match a target method. However, design patterns are not defined for detection or verification. Instead, they serve as guidance for various reification. Such sequence matching techniques can be limited in recognizing more common implementation variants.

Traditional data-flow analysis analyzes the entire AST of the method body. However, each behavior-driven pattern has a unique behavior that defines a target variable or statement for detection. To determine if an implementation is a correct pattern instance, we only need to verify whether the target does the right thing under the right condition. For example, if the `getInstance()` method of the Singleton pattern implements lazy instantiation, then it guarantees that the singleton instance gets created only once upon initialization. Here only the sub-AST that covers the lazy-instantiation mechanism requires full data-flow analysis.

Therefore, our approach uses data-flow analysis on ASTs in terms of basic blocks. As it processes each method body, it identifies the basic blocks, each of which contains statements that are executed under the same condition(s). Our approach links together the basic blocks based on execution flow to form a control-flow graph (CFG) for the method body. To illustrate, we present two examples: the Singleton and Flyweight patterns.

### **Example: the Singleton Pattern**

Consider our static behavioral analysis to determine lazy instantiation in a method body of `getTheSpoon()` in Figure 2.1. First, we build the CFG shown in Figure 4.1. (Control flow is indicated through directed edges.) Then, the CFG is scanned to determine which basic block instantiates and which returns the singleton instance. The main actor here is the singleton variable that has the roles of being instantiated and returned. Based on the

Figure 4.1: CFG of `getTheSpoon()` from Figure 2.1

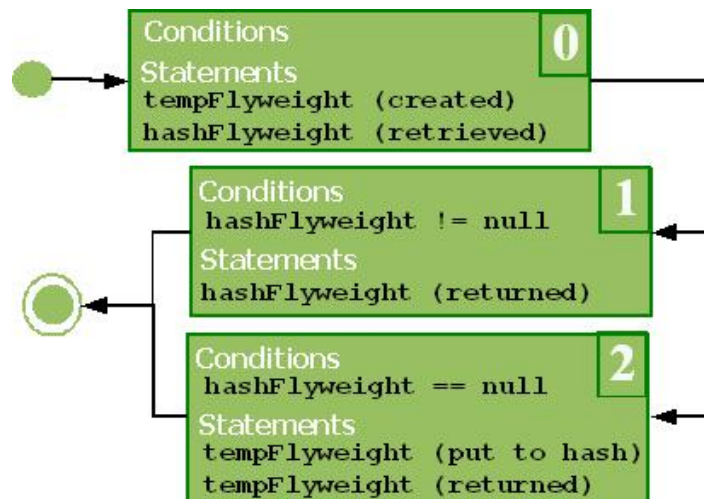
pre-determined actor and roles, our algorithm tells us `BasicBlock0` creates the singleton instance and `BasicBlock1` returns the singleton variable.

Then, we examine the conditions guarding `BasicBlock0`. Since only the last program state before return matters, here we use backward data-flow analysis on the flag variables involved in the conditions to verify if the contained sequence of statements guarantee single entrance to this basic block. Next, we check for the rest of the basic blocks if the flag variables can be modified, using backward data-flow analysis, elsewhere besides `BasicBlock0`.

Lazy instantiation can take many different forms. For example, one may use boolean types as flags, or use a different program structure, such as reversing the create-and-then-return order (shown in Figure 2.2). Such realistic variants of code map to (structurally) the same CFG as that in Figure 4.1, so we can use the same algorithm to track variable activities. Other behavior-driven patterns can also be detected using similar approaches.

### Example: the Flyweight Pattern

Our approach to detecting the Flyweight pattern is based on a similar technique, which analyzes the method that potentially returns a flyweight object. For example, consider the code in Figure 2.3. Our inter-class analysis pinpoints that `getFlyweight(...)` is a candidate method that returns a flyweight object; then our static behavioral analysis is applied to this method. We build the CFG shown in Figure 4.2 (which shows roles, described later). Similar to detecting the Singleton pattern, the actors, which are the flyweight instance and flyweight pool, and their roles must then be determined. In this case, a flyweight instance

Figure 4.2: CFG of `getFlyweight()` from Figure 2.3

is determined at a return statement. A flyweight pool is indicated by its data type, which is often a container class (such as a hashtable in this case). Based on the actors and their use and interaction in a statement, our algorithm assigns a role to each flyweight instance. Using backward analysis on this CFG, we can easily verify that the implementation either returns an existing or a new flyweight object.

Transforming to basic blocks not only flattens an AST of a method body, but also facilitates the detection of whether a target statement is executed in all paths. For example, the similarity between the CoR (with chained handlers) and the Decorator (with linked decorators) patterns is that each invokes the same polymorphic method of the adjacent node, but the difference is that this call is conditional for CoR and mandatory for Decorator. The same technique also applies to detection of loops to distinguish between the Observer and Mediator patterns.

While this lightweight approach is simple and is able to detect most common pattern implementations, it is limited in detecting more complicated implementations. The drawbacks of this method include: limited aliasing, lack of path sensitive analysis, limited con-

control over variable types<sup>1</sup>, lack of a full-blown inter-procedural analysis, and limited loop analysis.

---

<sup>1</sup>Currently, for detecting the control statement in the Singleton pattern, our method only targets booleans and `java.lang.Objects`, but not integer values that can also serve as control values)

# Chapter 5

## PINOT

Based on our methodology (Section 4), we implemented a fully automated pattern detection tool, called PINOT (Pattern INference recOverY Tool). The current implementation of PINOT recognizes all the GoF patterns in the structure- and behavior-driven categories.

### 5.1 Implementation

PINOT is built from Jikes (an open source Java compiler written in C++) with an embedded pattern analysis engine. There are number of advantages of using a compiler as the basis of a pattern detection tool. A compiler constructs symbol tables and AST that facilitate the inter-class and static behavioral analyses. Compilers also perform some semantic checks that help pattern analysis. For example, Jikes prints out warnings when a local variable shadows (has the same name as) a global variable, which helps disambiguate delegation relationships. Most importantly, compilation errors reflect the incompleteness of symbol tables and AST, which result in incorrect pattern detection results. However, some tools, such as FUJABA and PTIDEJ, are able to partially (with a fuzzy number) detect patterns from incomplete source. Such tools can be desirable if pattern detection is used as part of software forward-engineering, such as building and incorporating patterns on the run. In our case, pattern detection is reserved for reverse-engineering, where accuracy is



vital.

PINOT begins its detection process for a given pattern based on what is most likely to be most effective in identifying that pattern (i.e., declarations, associations, or delegations). This reduces the search space by pruning the least likely classes or methods. The completeness of a pattern detection tool is determined by the ability of recognizing pattern implementation variants. For practical reasons, PINOT focuses on detecting common implementation variants used in practice. Thus, some behavioral analysis techniques are not fully applied to each behavior-driven pattern. As an example, data-flow analysis is applied to analyzing the activities of the flag variable that guards the lazy instantiation in the Singleton pattern. The flag can have any data type, but `java.lang.Object` (when the reference for the Singleton instance also acts as the flag) and `boolean` are more common. Although a flag may be an integer, it is not as common in this case and would require much more computation. Thus, PINOT only analyzes lazy instantiation that uses `boolean` or `java.lang.Object` types. Inter-procedural data-flow and alias analyses are only used for detecting patterns that often involve method delegations in practice, such as Abstract Factory, Factory Method, Strategy and State patterns.

Some patterns, such as Decorator, CoR, Observer, and Mediator patterns, require only identifying the condition of which the target method delegation statement takes place. In particular, the Observer pattern involves a subject notifying a list of listeners. In Java, the listeners are usually stored in an array or a `java.util.Collection` class. If the latter, the iteration is often handled using `java.util.Iterator`. PINOT identifies arrays and array indexing, as well as classes that implements `java.util.Collection` and their use of `java.util.Iterator`. PINOT does not recognize any user-defined or user-extended data structures.

## 5.2 Results

We compared PINOT with two other similar tools: HEDGEHOG [28] and FUJABA 4.3.1 (with Inference Engine version 2.1).

HEDGEHOG is a pattern verification tool for Java. It defines a Prolog-like pattern specification language, called SPINE, for users to define their own pattern definitions. HEDGEHOG (see Section 2.1.2) reads pattern specifications from SPINE, which allows users to specify inter-class relationships and other path-insensitive semantic analysis (e.g., for Factory Method pattern, the predicate “instantiates(M, T)” checks whether a method M creates and returns an instance of type T.), but other more complicated semantic analysis is hard-wired to its built-in predicates (e.g., “lazyInstantiates(...”). Thus, SPINE is bounded by the capability of semantic analysis provided by HEDGEHOG. To use the tool, the user specifies a target class and a target pattern to verify against (i.e., attempt to recognize).

FUJABA has a rich GUI for software re-engineering. Its pattern inference engine provides a UML-like visual language for user-defined patterns. The language allows specifying inter-class relationships and a “creates” relationship (which is the same as the “instantiates” predicate defined in SPINE). FUJABA is easy to use: the user simply specifies the location of the source code and then runs the pattern inference engine. FUJABA displays the results graphically. FUJABA can run entirely automatically or incorporate interactive user guidance to reduce its search space.

PINOT is fully automated; it takes a source package and detects the pattern instances. All detection algorithms are currently hard-coded to prove the correctness of our techniques on the structure- and behavior-driven patterns.

Although these three tools were built for different uses, they all involve pattern recognition. Thus, we compare these tools in terms of accuracy. Table 5.1 shows the results of testing each tool against the demo source from “Applied Java Patterns”(AJP) [63]. Each AJP pattern example is similar to the one illustrated in the GoF book [38], except for the Flyweight pattern. The AJP Flyweight example does not define a flyweight pool; instead,

	Tools		
	PINOT	HEDGEHOG	FUJABA
<b>Creational</b>			
Abstract Factory <sup>†</sup>	✓	✓	×
Builder	–	–	–
Factory Method <sup>†</sup>	✓	✓	×
Prototype	–	×	–
Singleton <sup>†</sup>	✓	✓	✓
<b>Structural</b>			
Adapter <sup>*</sup>	✓	✓	×
Bridge <sup>*</sup>	✓	✓	✓
Composite <sup>*</sup>	✓	✓	×
Decorator <sup>†</sup>	✓	✓	×
Facade <sup>*</sup>	✓	–	✓
Flyweight <sup>†</sup>	✓	✓	×
Proxy <sup>*</sup>	✓	✓	–
<b>Behavioral</b>			
CoR <sup>†</sup>	✓	–	×
Command	–	–	–
Interpreter	–	–	–
Iterator	–	✓	×
Mediator <sup>†</sup>	✓	–	×
Memento	–	–	×
Observer <sup>†</sup>	✓	✓	×
State <sup>†</sup>	✓	×	–
Strategy <sup>†</sup>	✓	✓	✓
TemplateMethod <sup>*</sup>	✓	✓	✓
Visitor <sup>*</sup>	✓	✓	–

\*: a Structure-driven Pattern; †: a Behavior-driven Pattern

- ✓ the tool claims to recognize this pattern and is able to correctly identify it in the AJP example.
- ×
- the tool excludes recognition for this pattern.

Table 5.1: Pattern Recovery Results on AJP

the flyweight objects are statically instantiated and are static-final fields of the flyweight factory class. Table 5.1 shows that PINOT is able to recognize all the structure- and behavior-driven patterns in AJP. Because PINOT is a pattern detection tool, it assumes a class can participate in any pattern. Thus, PINOT tests a class against all pattern definitions. FUJABA was also tested in the same fashion. HEDGEHOG, however, is not an automated

verification tool and users are responsible of picking the patterns to verify against the target class. Thus, HEDGEHOG's results shown in Table 5.1 were based on prior knowledge of the source and only likely patterns were verified against a class [28].

Patterns can have various reification, and it is impossible for a pattern recognition tool to be complete. Thus, a tool's pattern-recognition ability depends on its interpretation of pattern implementation. As an example, the Observer pattern defines how a Subject class notifies its Listener classes. FUJABA recognizes a variant of this pattern and calls it the "Broadcast Mediator" pattern. It specifies that Subject has a container class for the Listeners, and there exists a method delegations from Subject to Listener. HEDGEHOG, on the other hand, first checks for a container (as does FUJABA) and then checks if Subject defines the following methods: a method that starts with prefix name "add", another that starts with "remove", and finally one method delegation that invokes some method in Listener. HEDGEHOG checks if the first two methods actually *add* and *remove* an object of Listener type from the container [27]. However, FUJABA's and HEDGEHOG's approaches do not capture the real intent of the pattern, which is the "broadcasting of notifications" as in a push-model communication. PINOT recognizes this intent by first identifying a container in a Subject class (based on inter-class relationships) and then using static behavioral analysis (using techniques similar to those illustrated in Section 4.2) to identify a loop control (e.g, in a notify method) that iterates through the container and invokes the same method (e.g., in an update method) of each contained Listener object.

We also tested PINOT on several real Java applications. Figure 5.1 shows only the results for Java AWT 1.3, JHotDraw 6.0 [10], Java Swing 1.4, and Apache Ant 1.6; see [20] for results on other applications, such as javac, java.io, and java.net packages. PINOT analyzes all classes, including anonymous and inner classes. A pattern instance is a collection of participating classes, and a class may participate in several other patterns. For example, in AWT, `java.awt.Component` and `java.awt.ComponentPeer` form one Bridge pattern instance; `java.awt.Component` and `java.awt.Container` together form one Composite and one

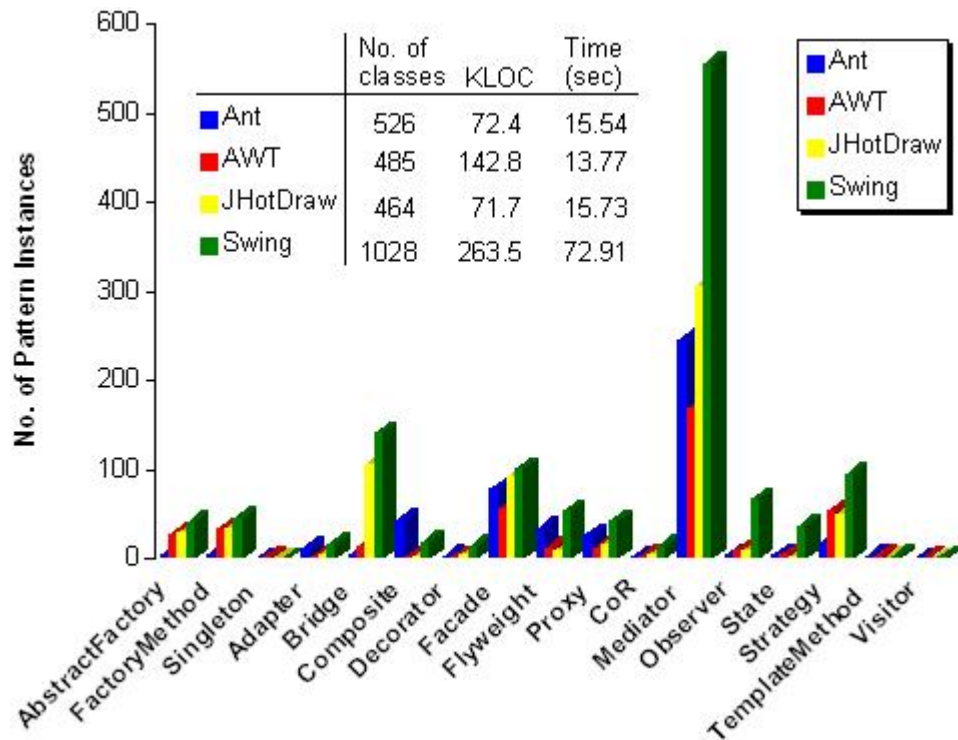


Figure 5.1: Pattern Instances Recovered

CoR pattern instances.

We ran PINOT on each of these packages on a Linux machine running on a 3GHz Intel processor with 1G of RAM. The timing results were promising. Compared to times for PINOT (Figure 5.1), PINOT took less than two minutes to analyze the entire Swing package, (with over a thousand classes) while FUJABA took 22 minutes to analyze the AWT and PTIDEJ took 2-3 hours to analyze JHotDraw. FUJABA was tested on a Pentium III 933MHz processor with 1G of memory. The reported time excludes parsing [56], but we are not certain if this time includes displaying the results graphically. PTIDEJ was tested on an AMD Athlon 2GHz 64b processor. PINOT is faster because the recognition algorithms are hard-coded and optimized to avoid redundant computation.

The PINOT website [20] comprehensively discusses the recovered pattern instances. Our test results were verified against an authoritative discussion pattern discussion board [18], documentation written by original developers [37], and manual verification. We found

some false positives in PINOT’s results: 23.75% of Factory Method instances are considered Prototype instances, of which the classes implement `java.util.Cloneable` and override the clone method. Such Prototype instances are trivial to identify using keyword matching. However, user-defined variants that do not implement the Java built-in types may require heuristics to verify the “cloning” intent within method bodies.

Due to the impreciseness of some GoF definitions (as discussed in Section 2.2), PINOT recognizes other common implementation variants of the Flyweight and Mediator patterns. In particular, PINOT recognizes *immutable* classes as a common implementation variant of the Flyweight pattern [27]. We found 13.69% of Flyweight instances as *immutable* classes. Moreover, PINOT detects a Mediator variant (in AJP and GoF sample code) that allows colleagues to be individual instances in a Mediator class (i.e., a variant 1:N relation). In this case, the Mediator class serves as a facade that shields direct communication from one colleague to another. We found 24.93% of the Mediator classes as Facade classes.

Unfortunately, we are not able to compare our results with other pattern recognition tools. HEDGEHOG verified 5 correct pattern instances [27] (that have also been identified by PINOT, see [20]) within the AWT, but the tool is not publicly available (unlike PTIDEJ and FUJABA). PTIDEJ [40] analyzes patterns at the bytecode-level and was tested on AWT and JHotDraw, but the results were not comprehensive and only presented recall results for the Composite pattern. FUJABA [56, 57, 60] was tested on the entire AWT 1.3, but only 3 pattern instances were reported (also identified by PINOT) and it is not clear whether the published results of pattern instances were comprehensive. Our experimentation with PTIDEJ and FUJABA indicates that PTIDEJ is not stable and lacks user documentation, while FUJABA works on small programs but has limited pattern recognition capability on larger programs. Reference [64] takes approximately 2.64 seconds (CPU time) on an Athlon XP 1400 MHz CPU with 1 GB RAM to analyze JHotDraw. The timing results include preprocessing and the actual pattern detection, while PINOT’s timing results also include I/O reporting times. Direct comparisons with Reference [64] cannot be

easily made, because of the differences in search criteria and the lack of detailed reporting. Reference [64] analyzes patterns based on only the structural constraints; no semantic analyses are involved. Further, Reference [64] reports the number of patterns instances found in JHotDraw, but the paper does not discuss how each pattern instance is formed (i.e., illustrating the class participants and their inter-class relationships) in the source code.

### 5.3 Discussion

So far, we have discussed the state-of-the-art pattern detection tools. Our contributions include: reclassifying the GoF patterns to facilitate pattern recognition; claiming that pattern definitions are either driven by code structure or system behavior; using our lightweight static program analysis techniques to efficiently recognize complicated program behavior; and implementing PINOT, a fully automated pattern detection tool that is faster, more accurate, and more comprehensive than existing tools. Our future work with PINOT includes: upgrading PINOT to recognize the latest version of the Java language, extending PINOT's recognition capability, and providing PINOT as a plugin to IDEs.

Currently, PINOT only analyzes source code written in Java 1.4. It is vital for PINOT to catch up with the new language constructs provided in Java (e.g., generics), as well as the latest JDK. Since PINOT was built by directly modifying the Jikes compiler, it further complicates the task of upgrading PINOT<sup>1</sup>. There have been discussions [12] on exposing the AST of the source code through the Java API. Once this API becomes available, the upgrading task will be much more feasible. Another option while this API is not yet available is to transform PINOT into an IDE plugin. Every IDE exposes its internal code structure to allow developers to write plugins. This way, as a plugin, PINOT is instantly upgraded as the IDE evolves.

PINOT's recognition capability is limited to the detectable GoF patterns (as discussed

---

<sup>1</sup>Our research group has initiated project **JPINOT** [20] that replicates PINOT on top of javac. This project serves as a foundation to facilitate future upgrading of PINOT.

in Section 5). We want to extend PINOT to recognize more complicated user-defined data structures and explore its use to detect design patterns in specific application domains, such as concurrent and real-time patterns. We also want to experiment with PINOT's use in tracking software evolution by design (as a comparison to References [39, 50]). That is, we would like to include some interesting and useful Java micro-patterns in PINOT.

Finally, we want to provide PINOT plugins to Java IDEs, such as Eclipse [4], JDeveloper [15], etc. Currently, PINOT is able to generate pattern reports in the XMI format. This standard format allows users to view PINOT results in a UML editor<sup>2</sup> Our plugin will include the current viewing capability and provide a better user interface that allows users to navigate through the pattern results and to go back to the code segment in the code editor where the pattern is implemented. Our plugin will also provide more flexibility in terms of executing the pattern recognition process. We want to investigate how users would be likely to use PINOT within an IDE. For example, PINOT can report pattern instances given a list of Java files. However, within an IDE, it may be more desirable if PINOT can report whether a class participates in some design pattern whenever the user places the cursor at a class declaration. Another related future work is to investigate whether users may want to use PINOT in combination with other source code related tools (e.g., source code version control tools, profilers, refactoring tools, and other static analysis tools) to facilitate their programming tasks. By exploring these possibilities, we can make PINOT more useful in facilitating the software re-engineering process.

---

<sup>2</sup>Currently, we have only tested this on ArgoUML [1].



## Chapter 6

# Formalizing Design Patterns

PINOT itself is a useful tool that detects concrete GoF patterns (as discussed in Chapter 5) from Java source code and then generates reports for each detected pattern instance. However, the algorithms for pattern detection and report generation are hard-wired, which limits the scope of detectable pattern definitions and the versatility of manipulating search results. From a usability standpoint, users may be interested in finding one particular implementation of a GoF pattern, whereas PINOT reports all common implementation variants. Moreover, users may want to see usages of a combination of patterns, whereas PINOT reports individual pattern instances. Finally, users may simply want to define their own patterns. This triggers the need to formalize design patterns, that is to identify the fundamental elements constituting a design pattern and to design a language for users to define one.

Researchers formalize design patterns for various purposes, mainly for software modeling, code generation, and pattern detection. The language style varies from different purposes. Design patterns are more effectively illustrated in terms of a visual language for software modeling, which requires more communication on software design. Code generation requires more detailed specification on the actual runtime behavior, thus design patterns are then expressed in terms of a programming language for precise code gener-

ation. Pattern detection, however, requires clear definitions of the constraints and search criteria for a design pattern. Thus, design patterns for recognition are often described using constraint-based languages, such as prolog-like or query-based languages. In the following sections, we discuss the various formalisms classified in terms of application purpose.

## 6.1 Software Modeling

Design patterns are often used at the coding level of software development. However, since each design pattern in the GoF book is presented with loosely defined UML diagrams, many researchers consider design patterns as part of the design level and propose methods and techniques to model design patterns. In this section, we present several studies in this area: LePUS [13], DPML [52], and Reference [51].

LePUS is a formal visual modeling language for object-oriented design. LePUS is used to specify generic design motifs, such as design patterns and object-oriented frameworks, as well as for modeling specific programs in object-oriented languages (such as Java, Smalltalk, and C++). However, this approach is found to be deficient and highly complex for specifying compound patterns and higher order participants [51]. Reference [43] applies the LePUS concept into UML using collaboration diagrams to specify the relation among pattern participants. However, inheriting the drawbacks from LePUS, this approach falls short in defining certain pattern intent. Section 7.1 discusses the drawbacks of using the LePUS concept to build a language for pattern detection.

DPML [52] (Design Pattern Modeling Language) is a visual language for modeling design patterns and their instantiation into UML design models. DPML models design patterns as a collection of participants. In particular, it models dimensions and constraints that associate with the participants.

Reference [51] extends UML 1.5 to specify the structural properties of design patterns. Reference [51] claims that since other pattern formalism work targets a particular type

of automation for future CASE tools (e.g., pattern verification), these studies tend overly formalize design patterns making the pattern specification ambiguous and inextensible to specify compound patterns. Reference [51] defines the structural properties of pattern leitmotifs and models the leitmotif structure using meta-level collaborations and stereotypes.

## 6.2 Code Refactoring and Generation

Design patterns are served as implementation guidelines to design a more extensible and maintainable software architecture. The GoF book [38] illustrates sample code for each design pattern. Each sample code shows a template implementation for building the foundational elements for that specific pattern. Thus, some researchers [32, 45] propose techniques to generate such pattern templates and leave the application implementations to the software developers. Reference [32] implemented a GUI for users to select proper pattern implementation (with certain parameters for users to select the desired data structures, e.g., lists vs. hashtables); reference [45] uses AspectJ to refactor and weave existing Java source code to generate the solution. Another work [29] takes a step further and view design patterns as a starting point for software development.

LayOM [29] provides design patterns with a first-class implementation construct corresponding to the conceptual design entity a pattern represents. However, first-class design pattern implementations require a more advanced language model than the conventional object-oriented model. Thus, LayOM extends object-oriented languages by providing language support for representing design patterns. A LayOM class is similar to any object-oriented class. Besides fields and methods that can be declared in a class, a LayOM class contains *layers*, *states*, and *categories* to facilitate pattern implementation. A layer intercepts incoming messages to the class where it is defined. The interception logic is based on the state of the class/object and the category (i.e., type) of the sender. A layer can encapsulate another layer to form a composition of patterns. A LayOM program can be translated

```
class adapter
layers
adapt : Adapter(accept mess1 as newMessA,
                accept mess2, mess3 as newMessB);
inh : Inherit(Adaptee);
end; // class adapter
```

Figure 6.1: The Adapter Layer in LayOM [29] Representing the Adapter Pattern

to C++ code. Layers are used to generate boilerplate pattern implementation, while the real behavior exists in the fields and methods defined by the programmer.

Figure 6.1 shows the Adapter class as a layer that represents class adaptation. The **Adapter** layer translates a `mess1` into a `newMessA` message and a `mess2` or `mess3` message into a `newMessB` message. The methods `newMessA` and `newMessB` are presumably implemented by a class `Adaptee` and the **Inherit** layer will redirect these and other messages to the instance of class `Adaptee` that is contained within the layer. The layers of abstraction preserves the traceability problem that occurs when applying design patterns into source code. However, LayOM is limited to express complex pattern behavior that is neither stateful or exclusive but is embedded in a method body. For example, the Flyweight pattern defines a `getFlyweight(key)` method that creates and returns shared flyweight objects on the fly. LayOM is not able to express the logic of `getFlyweight(key)`, because it functions regardless of the flyweight factory's current state and the types of callers. However, reference [45] is able to use AspectJ to define an abstract flyweight factory *aspect* that manages creation of shared objects.

The concept of layers in LayOM is similar to AspectJ [2] that provides language constructs to crosscut existing source code for flexible software refactoring. However, the complexity of refactoring increases as the software evolves, because the software architecture becomes more complex and more complex to apply generic design patterns. Another issue with LayOM is that design patterns are design guidelines instead of basic building blocks for software architecture. That is, we can easily break a piece of software by functionality

into software component but not by design patterns. This is because a participating element (e.g., a class) in a design pattern may also participate in another design pattern with another set of elements. Further, it hasn't been proven if there exists a finite set of fundamental design patterns for any software architecture. Thus, a software architecture can be built without applying any known design patterns. In parallel with the work on code generation from design patterns is Model Driven Architecture (MDA) [14] that generates executable code from model designs. MDA focuses on code generation from UML diagrams, which cover many aspects of software architecture (e.g., static class/component structure and dynamic runtime behavior). Several MDA tools adopt the concept of Executable UML [54]: UNIMOD [21].

Reference [33] argues that previous pattern realization techniques lack the ability to generate adaptable code. They adopt the idea of metaprogramming and designed an informal pattern specification language (PSL) for programmers to define more precise pattern realization. built a prototype tool Pattern Wizard that uses meta-programming Pattern Wizard provides micro-patterns and *tricks* to facilitate code generation. Furthermore, PSL's level of preciseness is rather detailed to the programming level.

Languages in this category focus on the ability to morph and weave class definitions and their underlying object-oriented design. This is because software designs are not pattern-oriented.

### 6.3 Pattern Detection

Design patterns increase software extensibility and maintainability if developers know exact location to extend. As the software evolves, it becomes less obvious to spot and trace uses of design patterns. This triggers some researchers to look into the extraction of design patterns from existing software.

Some pattern detection tools provide language support for users to domain-specific

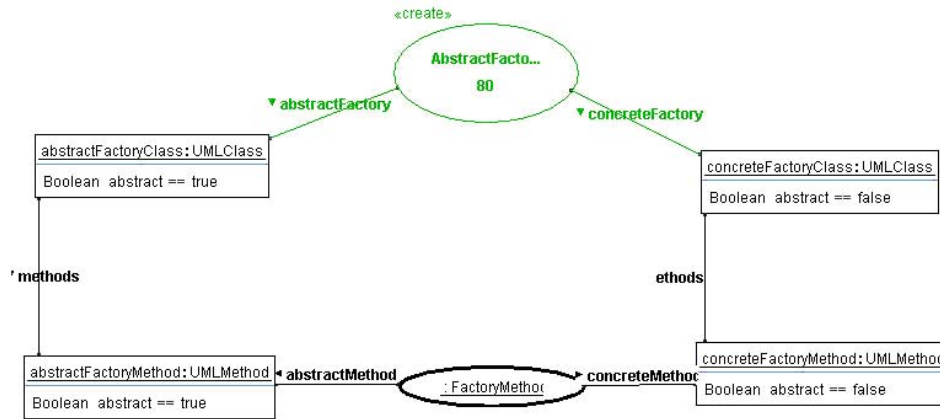


Figure 6.2: FUJABA's Specification for the Abstract Factory Pattern.

patterns [19]. The usefulness of a pattern language for reverse engineering depends on whether the language is able to (1) identify the basic building blocks (e.g., covering both the static and dynamic aspects of a pattern), and (2) provide the right set of language constructs. Over the past, researchers have proposed both graphical and textual notations for pattern specification. Graphical notations are generally based on UML, while textual notations vary from logic- and script-based languages.

FUJABA's pattern detection engine allows users to extend its pattern specification repository by providing a UML-like graphical language. FUJABA annotates UML entities (e.g., classes and methods) with additional *pattern* notations that represent inter-class constraints and association roles [6].

Figure 6.2 gives an example of specifying the Abstract Factory Pattern using FUJABA's annotated-UML. The squares represent UML entities each specified with a name with its entity type (a UML class or method) and a list of associated attributes. In this case, only the modifier "abstract" is used. The edges are directed and indicate ownership. Here it shows that each UML class owns a UML method. The ovals represent patterns, and each oval connects to its participants with specific participating roles. In this case, the specification illustrates that the Abstract Factory pattern is basically the Factory Method pattern but requires the parent factory class to be declared abstract. If the pattern is associated with

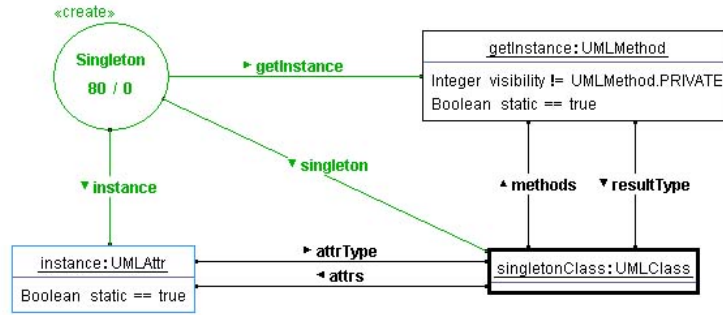


Figure 6.3: FUJABA’s Specification for the Singleton Pattern.

```

realises('AbstractFactory', [AF, AP]) :-
  forall(subclassesOf(AF),
    CF.exists(subclassesOf(AP),
      CP.realises('AbstractFactory', [AF, CF, AP, CP])
    )
  )
(* CF is a sub-type of AF and CP is a sub-type of AP,
  such that CF generates CP *)
realises('AbstractFactory', [AF, CF, AP, CP]) :-
  subtypeOf(CF, AF),
  subtypeOf(CP, AP),
  exists(methodsOf(AF),
    M1.and([
      typeOf(M1, AP),
      isAbstract(M1)
    ],
    exists(methodsOf(CF),
      M2.and([
        sameSignature(M1, M2),
        typeOf(M2, AP),
        instantiates(M2, CP),
      ]))
    ]))
  ]),

```

Figure 6.4: SPINE’s Specification for the Abstract Factory Pattern.

the text `<<create>>`, then it indicates that the pattern is detected if the participants are identified. The fuzzy numbers represent confidence and threshold to facilitate their pattern inference algorithm. FUJABA provides notations for specifying the structural requirements for class declarations and behavioral constraints for inter-class relationships. However, FUJABA lacks notations for specifying internal behavior within a method. For example, the Singleton pattern specification (shown in Figure 6.3) shows only the structural aspect of the pattern but lacks notations to specify how `getInstance()` controls object creation.

## Chapter 7

# A Comparison of Languages for Pattern Detection

In this chapter, we analyze three languages: the FUJABA pattern specification, SPINE, and LePUS used as pattern detection languages. We select these languages for comparison because each is complete enough to cover a fair amount of GoF pattern definitions. In Section 7.1, we compare the languages by their specification on the same patterns; in Section 7.2, we discuss the effectiveness of these languages.

### 7.1 Side-by-side Comparisons

These languages differ in purpose (software modeling vs. pattern detection) and style (visual vs. textual), and each language has its strength, as well as limitations. In this section, we compare pattern specifications that are available in all these languages and share identical pattern interpretations. For our comparisons, we selected two patterns that are commonly used and require precise specifications on both structural and behavioral aspects: the Abstract Factory pattern and the Observer pattern.



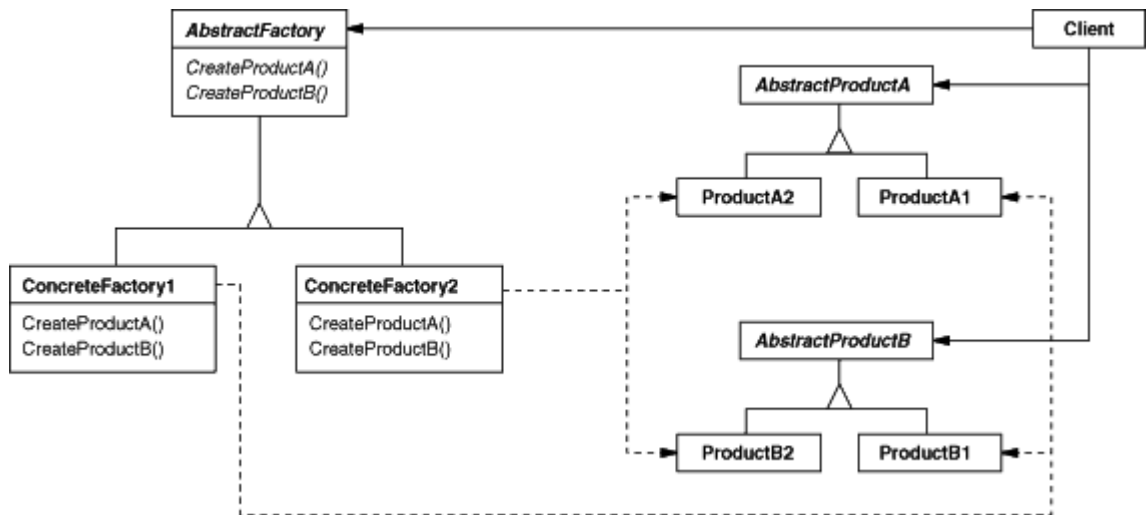


Figure 7.1: The OMT Specification for the Abstract Factory Pattern Specification [38]

### 7.1.1 The Abstract Factory Pattern

The Abstract Factory pattern involves two types of participants: the factories and the products, where each concrete factory creates its own concrete products. Figure 7.1 shows the UML class diagram for the pattern. Each participant is a separate class hierarchy and has an abstract class that defines a common interface for reification. To correctly define this pattern, a specification must illustrate separation of class hierarchies (for the structural aspect of the pattern) and object creation (for the behavioral aspect).

Figure 7.2 illustrates SPINE’s definition for the Abstract Factory pattern. Lines 2–5 define the separation of the factory and product class hierarchies. Lines 13–21 verify the object creation behavior by identifying the abstract and concrete factory methods. The `instantiates(M,C)` function verifies if method `M` creates and returns objects of type `CP`.

Figure 7.3 illustrates the pattern specified in LePUS, and Figure 7.4 shows the LePUS notations key. LePUS uses a triangle notation to express class hierarchies. The definition specifies two separate class hierarchies for factories and products, respectively. Object creation is defined using the `produce` association that connects the set of factory methods to the set of `Products`. The notations collectively specify that each factory method in a concrete factory class creates and returns some concrete product. LePUS has a notation

```

01 realises('AbstractFactory', [AF, AP]) :-
02   forall(subclassesOf(AF),
03     CF.exists(subclassesOf(AP),
04       CP.realises('AbstractFactory', [AF, CF, AP, CP])
05     )
06   )
07 (* CF is a sub-type of AF and CP is a sub-type of AP,
08   such that CF generates CP *)
09 realises('AbstractFactory', [AF, CF, AP, CP]) :-
10   subtypeOf(CF, AF),
11   subtypeOf(CP, AP),
12   exists(methodsOf(AF),
13     M1.and([
14       typeOf(M1, AP),
15       isAbstract(M1)
16     exists(methodsOf(CF),
17       M2.and([
18         sameSignature(M1, M2),
19         typeOf(M2, AP),
20         instantiates(M2, CP),
21       ]))
22     ])),

```

Figure 7.2: The Abstract Factory pattern in the SPINE Specification [27]

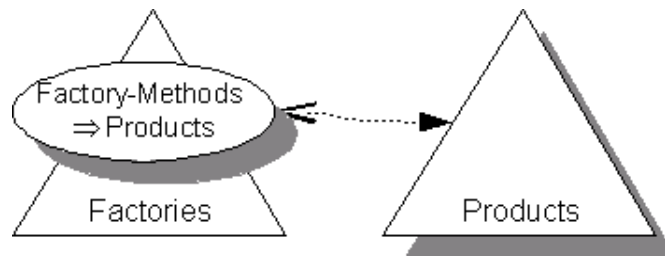


Figure 7.3: The Abstract Factory pattern in the LePUS Specification [13]

for `create`, which is similar to `produce` but without the line arrow on one side. Both `produce` and `create` indicate object creation, the difference is that `produce` further returns the created object.

Figure 7.5 illustrates FUJABA's specification. The specification shows only one class hierarchy for the factories. Object creation is defined in the subpattern "Factory Method", illustrated in Figure 7.6. The "Factory Method" subpattern specifies that an overriding method creates objects of type `productClass`. With Figures 7.5 and 7.6 combined, the factories and products are illustrated as different class hierarchies.

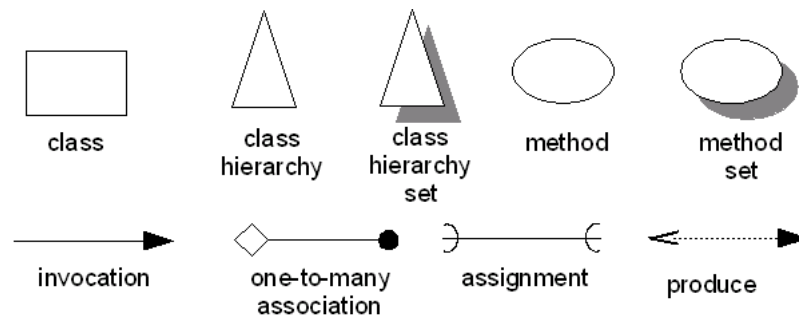


Figure 7.4: Partial LePUS Notation Keys [13]

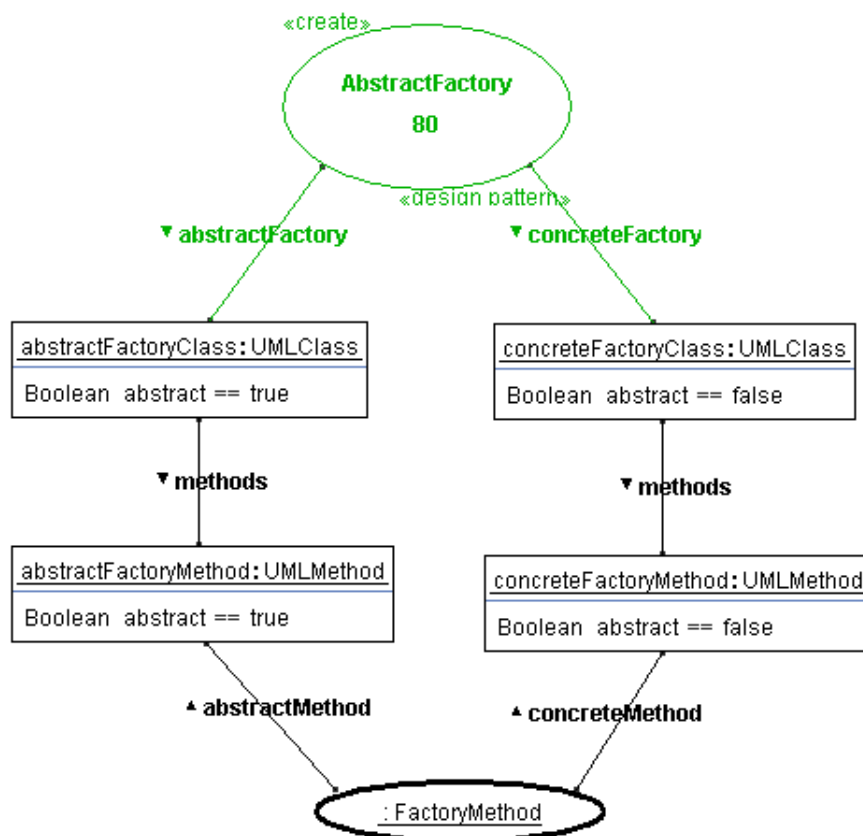


Figure 7.5: The Abstract Factory pattern in the FUJABA Specification [6]

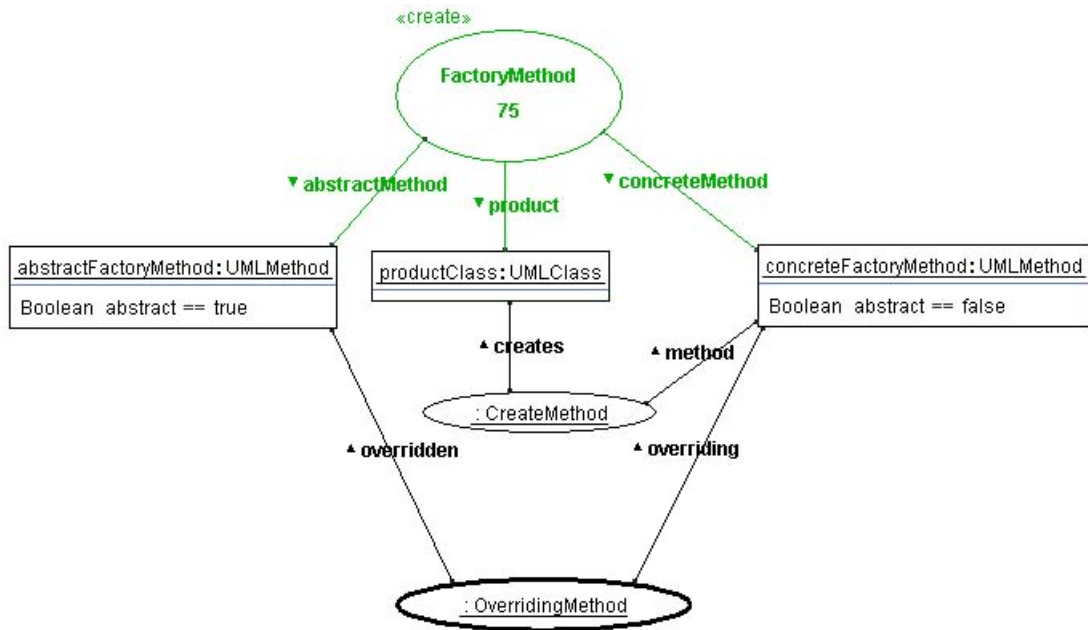


Figure 7.6: The Factory Method pattern in the FUJABA Specification [6]

These languages are able to clearly express both the structural and behavioral aspects of the pattern. LePUS takes a step further to distinguish `produce` and `create` emphasizing the duty of the factory methods. FUJABA, on the other hand, adopts a notation very similar to the OMT specification (shown in 7.1). Both the FUJABA and OMT specifications show reification of the concrete objects. FUJABA shows one for each participating class, while OMT shows two for each. Such specification somewhat indicates object quantity, which can be misleading and ambiguous for pattern detection.

### 7.1.2 The Observer Pattern

The Observer pattern involves two types of participants: the subject and the observers. Figure 7.7 shows the UML class diagram for the pattern. To correctly identify this pattern, a tool needs to detect the one-to-many structural and the notify-and-update behavioral relationships between the subject and observers.

Figure 7.1.2 shows SPINE’s definition for the Observer pattern. The structural relationship is established when the subject (which is the `Observable` in this definition)

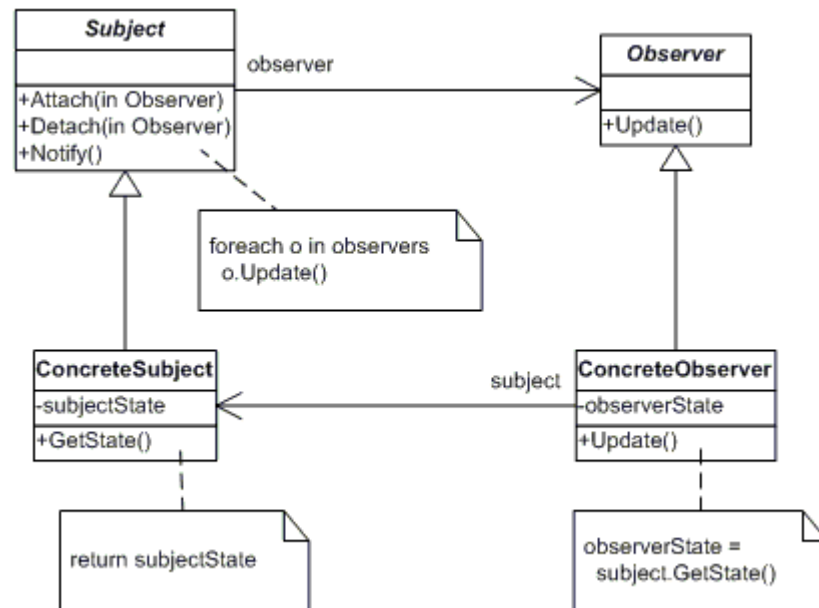


Figure 7.7: The OMT Specification for the Abstract Factory pattern specification [38]

```

01 realises (Observer, [Observable, Listener]) :-
02   navigable (Observable, Listener),
03   exists (methodsOf (Observable), M.and([
04     prefix (M, '' add '' ),
05     argsOf [M] = [A],
06     adds (M, A, Observable),
07     typeOf (A, Listener) ]),
08   exists (methodsOf (Observable), M.and([
09     prefix (M, '' remove '' ),
10     argsOf [M] = [A],
11     removes (M, A, Observable),
12     typeOf (A, Listener) ]),
13   exists (methodsOf (Observable),
14     M.exists (methodsOf (Listener),
15       D.invokes (M, D) ) ).
  
```

Figure 7.8: The Observer pattern in the SPINE Specification [27]

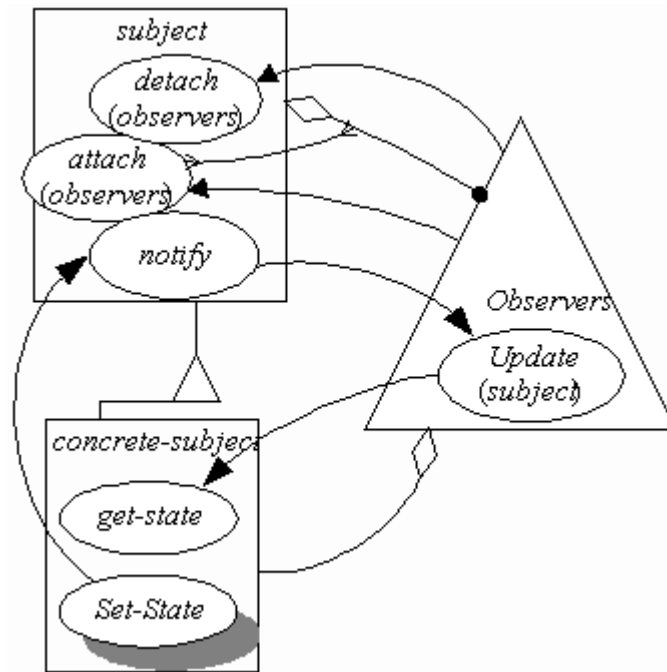


Figure 7.9: The Observer pattern in the LePUS Specification [13]

contains two methods each containing keywords `add` and `remove`, respectively. Since SPINE lacks predicates and functions that search for a one-to-many relationship [27], the definition suggests using keyword matching. The behavioral relationship is identified when there is a method in the subject class that invokes a method in the observer.

Figure 7.9 shows the Observer pattern specified using LePUS (refer to Figure 7.4 for the notations used in the specification). LePUS is able to effectively express the structural one-to-many association. Figure 7.9 further shows that this association is established by the method `attach`, and that the type of observers is not restricted to be homogeneous. The behavioral relationship is specified by the invocation from `notify` to `Update(subject)`. In addition to the notify-and-update relationship, the pattern specification (shown in Figure 7.9) requires that the observer retrieves the state of the concrete subject being passed in and then performs the update. The definition suggests that each observer can listen to multiple subjects. Further, it specifies that the set of `Set-State` methods (depicted in a shaded oval) trigger a `notify` event.

Figure 7.10 illustrates FUJABA's definition for the Broadcast Mediator pattern [6].

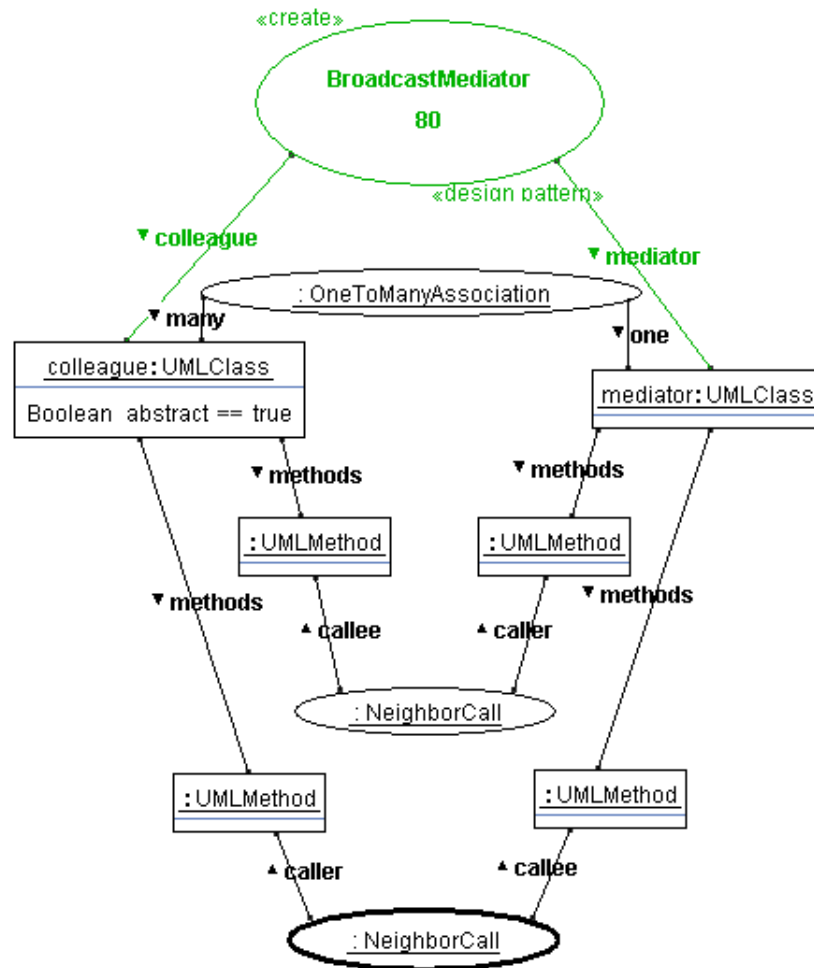


Figure 7.10: FUJABA's specification for the Observer Pattern [6]

The catalog of searchable GoF patterns of FUJABA Tool Suite 4.3.2 does not contain a pattern specification called the Observer pattern. However, the specification for the Broadcast Mediator pattern illustrates search criteria that are roughly equivalent to those for the Observer pattern. The structural relationship is specified through `:OneToManyAssociation`, which analyzes field references. The behavioral relationship is specified through two `:NeighborCall` notations from the subject (i.e., the mediator) and to the observer (i.e., the colleague), and vice versa.

For the structural one-to-many relationship, LePUS and FUJABA are more explicit requiring the association to be established through field references; SPINE is more implicit requiring certain collection operation methods to be available. As for the behavioral notify-and-update relationship, none of the specifications are able to express the iteration of notification to the observers.

## 7.2 Discussion

Each language is different in its purpose and style. These differences result in different specifications for the same design pattern. FUJABA and SPINE are used to detect patterns in Java source, thus their languages allow users to specify more detailed code properties, such as modifiers, visibilities, ownerships for classes, methods, and fields. LePUS, on the other hand, is designed for software modeling, and its language constructs focus on higher-level inter-class relationships.

As it turns out, none of these languages are able to specify all 23 GoF patterns. SPINE's builtin predicate set only works at a weak semantic level, and it cannot express specific behavior that is dependent on how (and where) it is used, or has dependencies on the semantic interpretation of methods. Such patterns include: the Facade, Builder, Command, CoR, Interpreter, Mediator, Memento patterns [27]. LePUS, on the other hand, is not bound by program semantics, it is able to express a broader scope of patterns. LePUS focuses on



inter-class relationships, but lack notations for specifying intra-class behavior (e.g., the Singleton pattern). Further, LePUS also lacks notations for expressing patterns that can only be defined with specific inter-class behavior( i.e., domain-specific), such as the Prototype, Mediator, Interpreter, Memento patterns. FUJABA tries to express patterns based on the UML class diagram with several basic behavioral relationships, such as **create** and **delegate**. Without incorporating other behavioral UML diagrams (such as statecharts, activity, and communication diagrams), FUJABA cannot express patterns that involve other more specific behavioral relationships such as the assignment association defined in LePUS (see Figure 7.4) that can conveniently distinguish the State and Strategy patterns.

In terms of expressiveness, LePUS has the richest syntax to express complex inter-class relationships (such as class hierarchy, one-to-many aggregation, assignment association, etc.). LePUS also allows specification on method signatures. (Such a specification can be placed within the oval that LEPUS uses to denote a method.)

In terms of preciseness, SPINE is the most suitable language for pattern detection, because it facilitates specification on pattern criteria and even search steps. SPINE allows users to specify the declarations of fields, methods, and classes. The drawback of the language is the lack of built-in functions and predicates for pattern behavior, which limits language extensibility.

In terms of practicality, FUJABA adopts a UML-based notation and defines entities for classes, methods, and fields. These entities are connected through links combined with text to convey pattern behavior and participant roles. However, the language lacks notations to specify pattern behavior, which results in overly complex definition (e.g., using `:Neighbor-Call` and other entities combined to express a simple method invocation). Further, FUJABA specifies patterns at the class definition level, which can be misleading. For example, Figure 7.11 shows FUJABA's *Normal Mediator* pattern (which specifies the Mediator pattern in GoF). The specification is very similar to the OMT specification (shown in Figure 7.12). Both specifications illustrate two concrete colleagues. The pattern is defined as [38]:

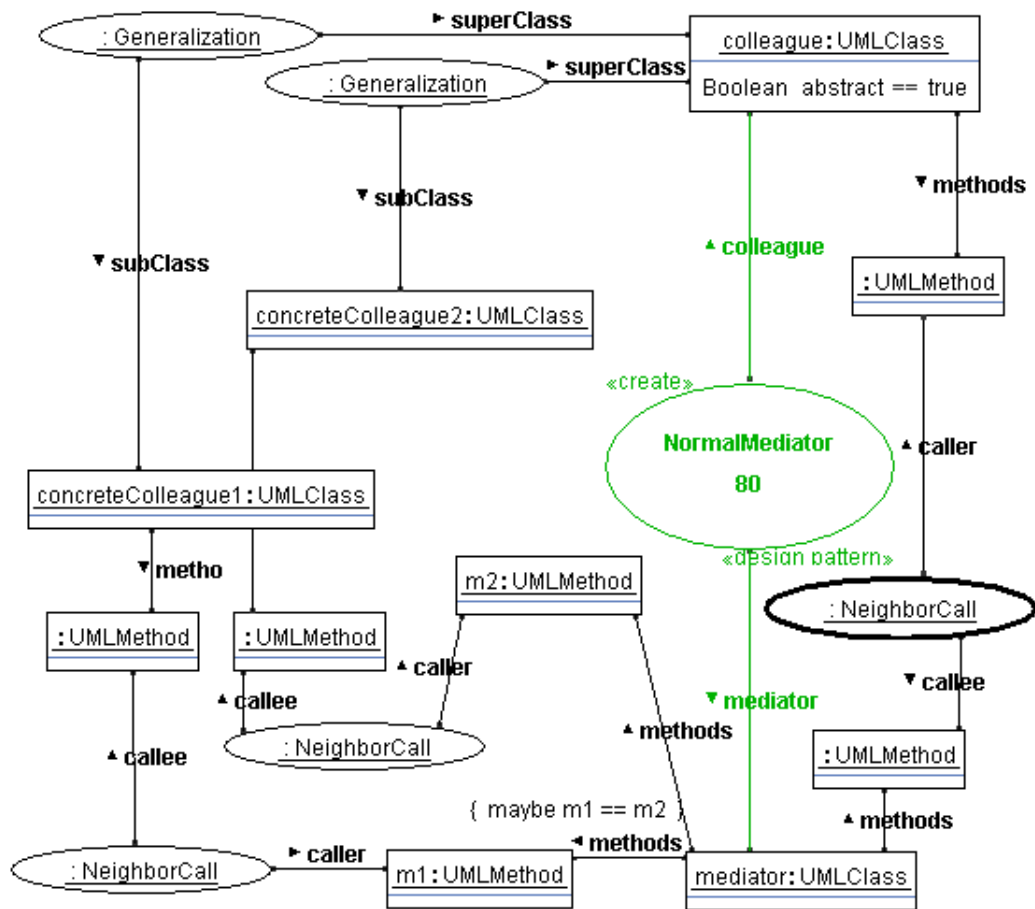


Figure 7.11: The Normal Mediator Pattern in FUJABA [6]

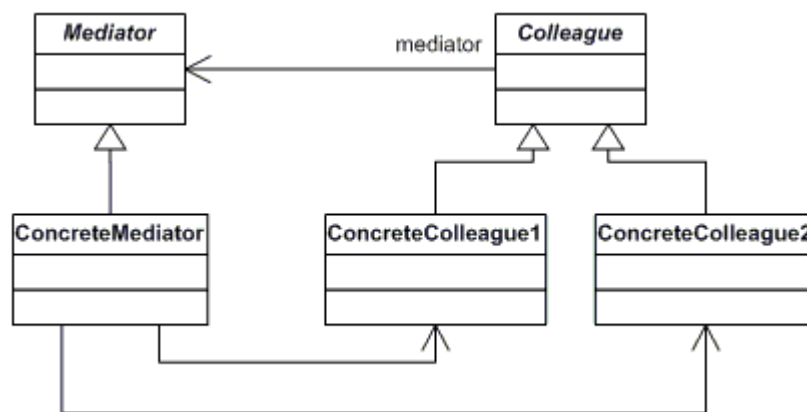


Figure 7.12: The OMT Specification of the Mediator Pattern [38]

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

We know that the two concrete colleagues only serve as an illustration instead of a criterion for detection. For pattern detection, we need a specification that is precise enough to avoid ambiguity, yet abstract enough to capture program intent.

We want to come up with a language for PINOT that is expressive enough to describe both the structural and behavioral aspects of a detectable pattern (described in Chapter 3); precise enough to distinguish between similar patterns (such as the State and Strategy patterns); and practical enough to use and to integrate with PINOT. The next chapter discusses our pattern detection language as well as its evaluation and performance results.

# Chapter 8

## MUSCAT

This chapter discusses the design and implementation of our pattern detection language, MUSCAT (**M**inimal **U**ML **S**pecific**C**ATion). MUSCAT is designed to make pattern design more effective, because the language itself is simpler and its implementation enhances usability. As part of the implementation of MUSCAT, we extend PINOT by providing a pattern detection API, which we will refer to as the PINOT API. Pattern specifications defined in MUSCAT are translated to C++ source code that uses the PINOT API. The user then compiles and executes the the C++ code (which is linked with the PINOT API) to detect the patterns they defined. Section 8.1 discusses the guidelines we adopt to design MUSCAT. Section 8.2 illustrates MUSCAT’s language constructs. Section 8.3 shows PINOT’s pattern detection API. Section 8.4 describes the MUSCAT implementation. Finally, Section 8.5 evaluates MUSCAT based on performance, accuracy, and completeness.

### 8.1 Design Guidelines

Based on the comparison analysis in Chapter 7, we feel that it is vital for a pattern detection language to be both simple and usable. Thus, we use language **simplicity** and **usability** as the two principal guidelines to design MUSCAT.

Architectural designs (usually defined in some modeling language or other forms of

documentation) are usually passed among developers as a communication method to convey their ideas. We want each user-defined pattern specification alone to be clear and easy enough to convey the details of an architectural design. Thus, we define language **simplicity** as the ability for users to quickly grasp the design concepts from a pattern specification. Visual languages use a combination of shapes, lines, and text to guide readers through the idea of the participating entities and their relationships in an architectural design. As illustrated in Chapter 7, both LePUS and FUJABA are visual languages and have the advantage of facilitating readers' understanding more effectively than the text-based language SPINE. Another aspect of language simplicity is related to its syntax and semantics. As a software system evolves, its internal design becomes more complicated to describe. We need a balance between having a variety of language constructs and the level of detail of which the language can describe. As an example, LePUS and SPINE are syntactically richer than FUJABA, since LePUS and SPINE each uses more visual constructs and predefined predicates and functions, respectively. However, the amount of the language constructs may introduce ambiguity (if lacking) or complexity (if overwhelming) in terms of understanding and specifying pattern definitions.

Then, we define **usability** as the overall user experience of the tool support from editing to pattern detection. SPINE is a Prolog-based language. Because it is text-based, users can use any text editor to define their specification. The source code is then processed by HEDGEHOG [27]. FUJABA is a UML-based language. The FUJABA Tool Suite RE is an IDE for Java, the IDE supports various re-engineering tools (such as the pattern inference engine). Users can edit and run pattern rules within the IDE. LePUS is a visual language. As illustrated in Reference [13], users can use Microsoft Visio to create a LePUS specification. However, there are no known pattern detection tools that support pattern specifications defined using LePUS.

Based on these guidelines, we decide to use the syntax from UML, a commonly used modeling language, but to simplify it specifically for defining design patterns. UML is

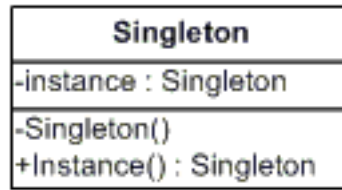


Figure 8.1: The UML Class Diagram for the Singleton Class [3]

widely used in software engineering. Its popularity motivated various tools support (such as editing, code generation, etc.) and format standardization (such as XMI, which is a commonly used as an interchange format for UML models). Thus, using UML, as opposed of other proprietary languages, greatly enhances usability. However, UML contains 13 types of diagrams (as in UML2.0) to describe various levels of system design. Using UML entirely for defining design patterns is overkill. A pattern definition contains definitions for its structural and behavioral aspects of the pattern. Then, strictly speaking, we need only a class diagram that defines the structural aspect and a statechart/activity diagram that defines the behavioral aspect. However, each diagram type is self-contained and lacks continuity between a higher-level diagram to a lower-level diagram. For example, a class diagram allows detail definitions for a field declaration and a method signature. To define the method body using a statechart that corresponds to a method declaration specified from a class diagram is not straightforward, because a statechart diagram is composed of states, transitions, and triggering events. Additional annotation will be required to bridge the two diagrams.

To simplify UML, we can use only the class diagram, that is to view a design pattern at the class level. Yet, using the class notation entirely introduces ambiguity, since it lacks notations to specify certain declaration constraints. For example, a method signature is defined with a proper type name and the exact parameter list. Its preciseness leaves no flexibility for defining generic declarations or specifying multiplicity of declarations. Figure 8.1 shows a UML class diagram for the Singleton pattern. However this definition

is ambiguous as follows: the public (denoted by +) method `Instance():Singleton` does not specify whether the constructor may have parameters or must not have parameters (as required); the only private (denoted by -) field `instance:Singleton` is significant and does not allow or prevent (as required) any more fields with the same declaration; the specification of the private constructor implies that all constructors in this Singleton class also be private. Therefore the class notation must be further simplified by removing field and method declarations within a class notation. In the next section, we illustrate how MUSCAT allows users to define a design pattern at the class level without losing specific implementation details.

## 8.2 MUSCAT Language Constructs

To view a design pattern at the class level, there are only three elements: the participating classes, inter-class associations, and pattern rules. Figure 8.2 illustrates the UML notations that MUSCAT uses for these language constructs. The class and inter-class association describes the structural aspect of a pattern. A pattern rule defines the behavioral constraints over the structural settings formed by the classes and inter-class associations. MUSCAT defines 13 pattern rules for both classes and inter-class associations. Each pattern rule describes a fundamental pattern constraint at the class level. Table 8.1 illustrates each pattern rule with its notation and usage.

## 8.3 PINOT API

As part of implementing MUSCAT, we extend PINOT by providing a set of pattern detection API. Table 8.2 illustrates the PINOT API. The API is designed to match MUSCAT's language constructs (discussed in Section 8.2). Thus, the API offers a higher-level (i.e., the class-level) accessibility to the internal structures of the AST and symbol tables

Pattern Rule	Description	GoF Example(s)
<<singleton>>	This rule applies to classes and specifies that a class behaves as a singleton class defined in the Singleton pattern.	Singleton
<<facade>>	This rule applies to classes and specifies that a class behaves as a facade class defined in the Facade pattern.	Facade
<<creates: <i>type</i> >>	This rule applies to directed associations and specifies that the left-hand-side class creates objects of type <i>type</i> , which is also the right-hand-side class.	Abstract Factory
<<!creates: <i>type</i> >>	This rule applies to aggregation associations and specifies that the left-hand-side class creates objects of type <i>type</i> , which is also the right-hand-side class. The difference from the previous creates pattern rule is that each object is created once and stored in the aggregation.	Flyweight
<<accepts( <i>type</i> )>>	This rule applies to directed associations and specifies that the left-hand-side class allows access from type <i>type</i> , which is also the right-hand-side class.	Visitor
<<delegates>>	This rule applies to aggregation associations and specifies a method delegation.	Strategy
<<!delegates>>	This rule applies to aggregation associations and specifies an exclusive method delegation.	Adapter Proxy Mediator
<< *delegates>>	This rule applies to aggregation associations and specifies a one-to-many method delegation (i.e., broadcasting).	Observer
<<delegates(decorates)>>	This rule applies to aggregation associations and specifies that the method delegation continues invoking the same method in its super class.	Decorator
<<delegates(condition)>>	This rule applies to aggregation associations and specifies that the method delegation occurs only conditionally.	CoR
<<!sets( <i>var</i> )>>	This rule applies to dependencies and specifies that the aggregation specified by <i>var</i> can be modified by the dependency's left-hand-side class.	State
<<independent>>	This rule applies to a bidirectional association and specifies that two class hierarchies are mutually independent.	Bridge

Table 8.1: Pattern Rules in MUSCAT









<b>Class</b>		A class notation. May be abstract. For specifying a participant.
<b>Inter-class Association</b>		An aggregation relation. May specify multiplicity. For specifying association by ownership.
		A directed association relation. For specifying association besides ownership. E.g., by parameters.
		A dependency relation. For specifying association by indirect reference (i.e., not through direct ownership). E.g., class A creates class B, thus B depends on A.
		An inheritance relation. For specifying association by class inheritance.
		A bidirectional association relation. For specifying association by class/object constraints. E.g., class A and B are not derived from the same class. Thus, A and B are associated by such a class constraint.
<b>Pattern Rule</b>	<code>&lt;&lt; rule &gt;&gt;</code>	A stereotype. For specifying pattern rules, used to describe a class or an inter-class association.

Figure 8.2: Language Constructs for MUSCAT

constructed by PINOT. There are tradeoffs between offering a low-level API or a high-level API. A lower-level API may allow users to access the details of each symbol and may include a subset of the API specifically for flow analysis. While a low-level API offers versatility, it complicates the task for code generation, as we find necessary for implementing MUSCAT. Although a high-level API may be less flexible depending on how much details a pattern specification requires, it is still important that the API is coherent (i.e., captures the fundamental conceptual elements at that level) and is powerful (i.e., recognizes most implementation variants). The next section explains how we generate the pattern detection code that uses the PINOT API from a MUSCAT specification.

<code>bool isAbstract (TypeSymbol *type)</code>
Returns true if <i>type</i> is an abstract class; otherwise false.
<code>bool isFacadeClass (TypeSymbol *type)</code>
Returns true if <i>type</i> is a facade class, based on the Facade pattern; otherwise false.
<code>bool isSingletonClass (SymbolTables *sym_tables, TypeSymbol *type)</code>
Returns true if <i>type</i> is a singleton class, based on the Singleton pattern; otherwise false.
<code>bool isFactoryMethod (SymbolTables *sym_tables, MethodSymbol *method)</code>
Returns true if <i>method</i> is a factory method, based on the Factory Method pattern; otherwise false.
<code>bool createsFlyweights (VariableSymbol *var, TypeSymbol *component_type)</code>
Based on the Flyweight pattern. A flyweight factory creates and returns a flyweight object of type <i>component_type</i> . These flyweight objects are stored in a flyweight pool, referenced by <i>var</i> . This method returns true if <i>var</i> and <i>component_type</i> participate in a Flyweight pattern.
<code>bool areIndependentHierarchies (SymbolTables *sym_tables, TypeSymbol *type1, TypeSymbol *type2)</code>
Returns true if <i>type1</i> and <i>type2</i> do not derive from the same root class (besides <code>java.lang.Object</code> ); otherwise false.
<code>VariableSymbol *isVisitorMethod (MethodSymbol *method)</code>
Based on the Visitor pattern. Returns true if <i>method</i> accepts a visitor class and invokes the corresponding visit method; otherwise false.
<code>bool star_delegates (SymbolTables *sym_tables, AstVariableDeclarator* var, TypeSymbol *type, TypeSymbol *component_type)</code>
Returns true if <i>type</i> and <i>component_type</i> form a one-to-many relation through <i>var</i> and there exists broadcasting delegations to each <i>component_type</i> object; otherwise false.
<code>bool bang_delegates (SymbolTables *sym_tables, TypeSymbol *left, TypeSymbol *right)</code>
Returns true if <i>left</i> has exclusive access to <i>right</i> ; otherwise false.
<code>bool cond_delegates (SymbolTables *sym_tables, VariableSymbol *var)</code>
Based on the CoR pattern. Returns true if <i>var</i> refers to the next handler in the chain and passes a request to this next handler if the current handler cannot handle it; otherwise false.
<code>bool decor_delegates (SymbolTables *sym_tables, VariableSymbol *var)</code>
Based on the Decorator pattern. Returns true if <i>var</i> refers to the successor object and invokes the same method defined in the successor.
<code>bool *bang_sets (SymbolTables *sym_tables, TypeSymbol *setter, VariableSymbol *var)</code>
Returns true if <i>setter</i> sets the value of <i>var</i> ; otherwise false.

Table 8.2: PINOT API

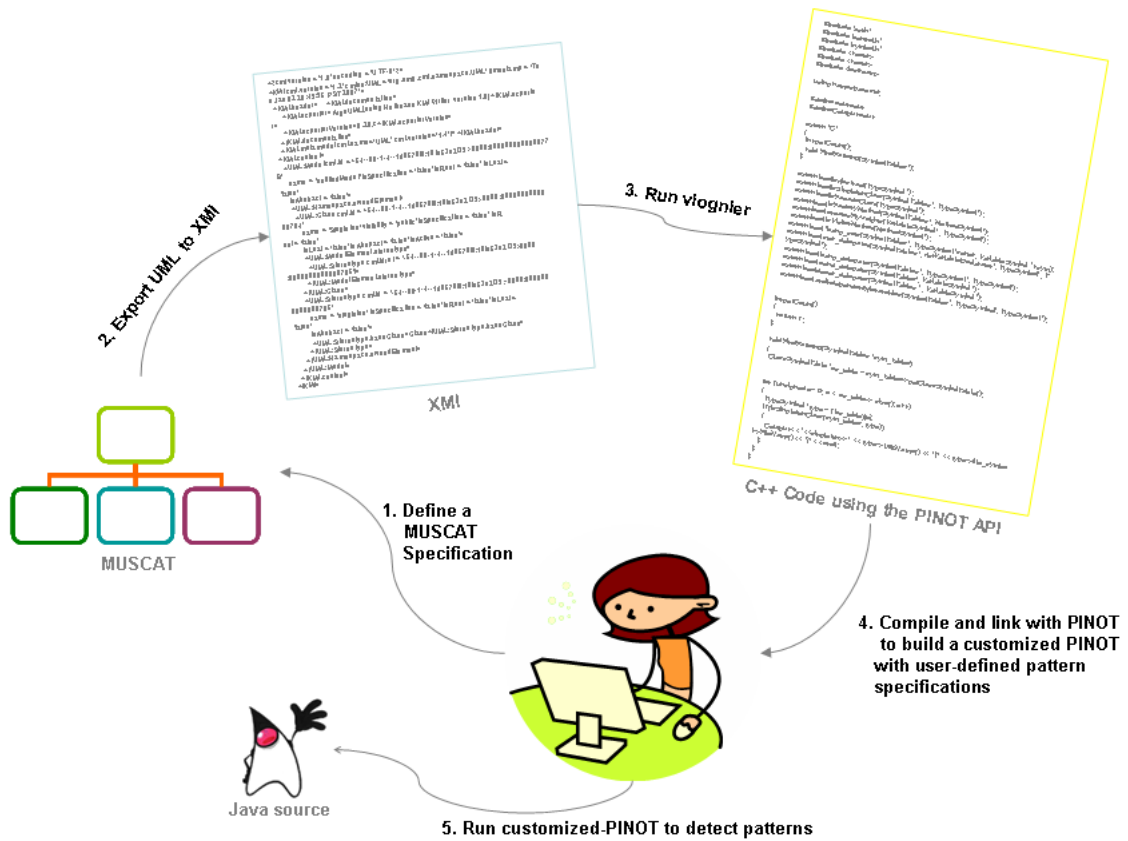


Figure 8.3: The Implementation of MUSCAT

## 8.4 Implementation

Figure 8.3 illustrates the implementation of MUSCAT. Since MUSCAT uses part of the UML language constructs, users can use any UML editors to edit their MUSCAT pattern specification. The only requirement is that the UML editor must be able to export diagrams to XMI format. In our experiment, we use ArgoUML [1] because it is open source and exports diagrams to a standard XMI format. Once an XMI file is available, we then generate C++ code that uses the PINOT API based on the information provided in the XMI. The translation and code generation is done by our Python script *viognier* (which we will discuss later). The user then compiles and executes the C++ code (which is linked with the PINOT API) to detect the patterns they define.

*Viognier* generates a complete C++ program by first generating the **#include** statements

and **external** function declarations that define the PINOT API. Then, *viognier* exposes two types of functions. The first is a single `int getCount()` function, which returns the total number of generated pattern detection functions. The second are multiple `void FindPattern $i$ (SymbolTables*)` functions, each of which defines an individual pattern detection function. Each pattern detection function corresponds to one XMI specification. The integer  $i$  is determined by the order in which an XMI file appears in the command line.

*Viognier* generates C++ code for each pattern detection function by first parsing its corresponding XMI file using an XML DOM parser, then collects the language elements, and finally generates its definitions. The language elements collected from the XMI file include: stereotypes, classes, associations, and dependencies. Each pattern detection function must have a starting point that drives the pattern detection logic. Pattern detection can start by traversing either the list of classes, fields, or methods. These symbols are available through the PINOT API, which exposes the internal class, field, and method symbols created by PINOT. Thus, a MUSCAT specification must have at least one class (which drives the search by traversing all classes), association (which drives the search by traversing all fields), and aggregation (which drives the search by traversing all methods) element. We call these elements the *driving* elements. The simplest diagram is a single class definition, such as a class defined with the `<<singleton>>` stereotype. In this case, *viognier* simply generates a function that traverses the list of available class symbols and passes each class symbol into the corresponding class constraint method, such as `bool isSingletonClass(...)`. As for a more common diagram, where inter-class associations are defined, *viognier* starts generating code from a directional association. If the directional association is an aggregation, then *viognier* generates code for identifying an aggregation between two classes. Then, based on these two classes, *viognier* generates constraints for each stereotype defined for the classes. Otherwise, if the directional association is an association, then *viognier* generates code for identifying the method that associates these two classes together and, similarly, generates constraints for each stereo-

type defined for the classes. The MUSCAT dependency, inheritance, and bidirectional association relations are constraint specifications that modify driving elements.

## 8.5 Evaluation

This section discusses how we evaluate the effectiveness of MUSCAT. We conduct two experiments. In the first experiment, we focus on evaluating correctness and speed. In this experiment, we try to define detectable GoF patterns (that are available in PINOT) using MUSCAT. Then, we use these specifications to construct a *customized* PINOT (as explained in Section 8.4 and illustrated in Figure 8.3) and compare correctness and speed between the *customized* PINOT (which we will refer to as cPINOT) versus the regular PINOT (which we will refer to as PINOT). In the second experiment, we evaluate usability. In this experiment, we try to define some commonly used design patterns that are not defined in the GoF book. Using these specifications, we also construct a *customized* pattern detection engine and run it again on the same benchmark applications used in our first experiment. Sections 8.5.1 and 8.5.2 illustrate our first and second experiments, respectively. The C++ code translated from the MUSCAT specifications illustrated in Sections 8.5.1 and 8.5.2 can be found in Appendices A and B, respectively.

### 8.5.1 Defining the GoF Patterns

In this experiment, we define most of the PINOT-detectable patterns except for the Factory Method and Template Method patterns. The Factory Method pattern can be described using the association notation. ArgoUML does not allow specifying only an association without specifying the associated classes. The Factory Method pattern specification is used in combination with other patterns, such as the Abstract Factory pattern. The Template Method pattern describes a design property that stresses how a method is declared and implemented. A *template* method is declared *final*, and it only calls the non-*final* methods

(which are the *hook* methods) within the owning class. ArgoUML allows specifying an operation (i.e., a method) with its signatures and choices of visibility, modifiers, and concurrency properties. Specifying modifiers is possible, but ArgoUML does not allow users to specify method behavior, e.g., implementation in the method body. While adding a stereotype for such method is feasible in ArgoUML, MUSCAT only supports stereotypes for classes and associations. However, in the future we may consider including such language construct in MUSCAT. In our *customized* PINOT, we include 15 MUSCAT specifications; each defines a detectable GoF pattern (except for the Factory Method and Template Method patterns).

### Specifications and Correctness

Both cPINOT and PINOT start their analyses by first building the AST and symbol tables and then run the pattern search routines<sup>1</sup>. The difference is that cPINOT runs each input pattern specification independently, while PINOT, for optimization purposes, may analyze similar patterns together (e.g., the State and Strategy patterns). This affects both results and timing (which will be discussed in Section 8.5.1). We can categorize, as shown in Figure 8.4, the causes of differing results from cPINOT and PINOT into: pattern interpretation, language deficiency, and pattern reporting.

### Singleton and Facade

The MUSCAT specifications used in cPINOT are illustrated in Figures 8.5 and 8.6. Both the *customized* and the regular PINOTs agree on each instance of these patterns. This is because the PINOT API exposes the exact same criteria for analyzing these two patterns. Figures A.13 and A.8 show the C++ code translated from the MUSCAT specifications for the Singleton and Facade patterns, respectively.

---

<sup>1</sup>For cPINOT, these search routines are translated from the MUSCAT specifications.

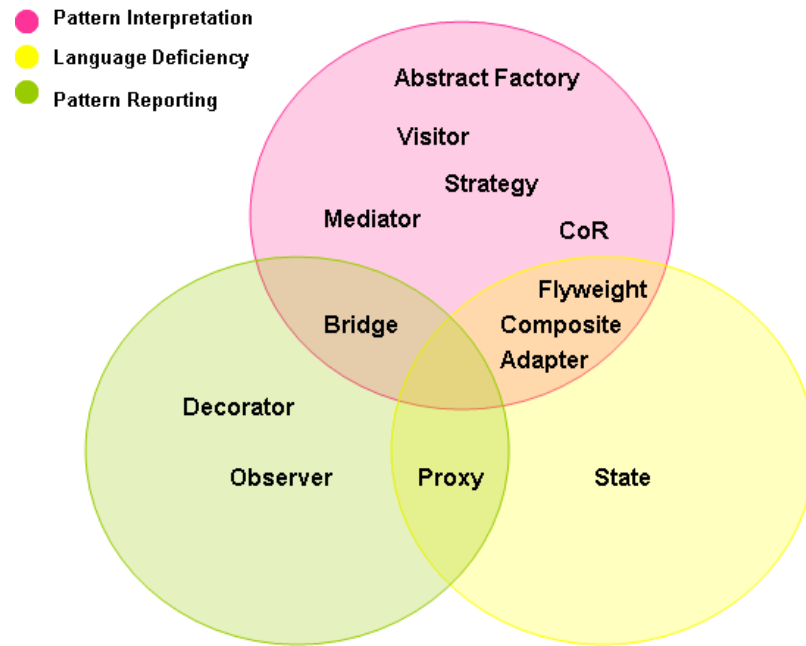


Figure 8.4: Categorization of the Differing Results

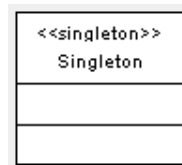


Figure 8.5: The MUSCAT Specification on the Singleton Pattern

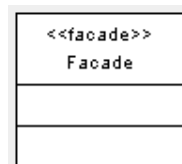


Figure 8.6: The MUSCAT Specification on the Facade Pattern

### Abstract Factory

Table 8.3 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.7 (where its translated C++ code is illustrated in Figure A.2). Different from PINOT's search criteria is that the MUSCAT specification requires both the factory and product classes to be declared abstract. This constraint can be specified in MUSCAT.

	Ant	AWT	JHotDraw	Swing
PINOT	6	28	30	41
cPINOT	0	0	0	0

Table 8.3: Comparisons on the Abstract Factory Pattern

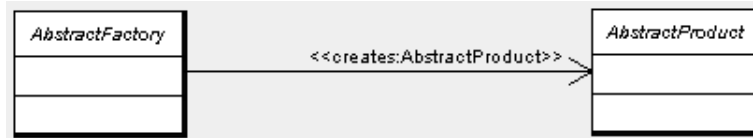


Figure 8.7: The MUSCAT Specification on the Abstract Factory Pattern

**Visitor**

Table 8.4 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.8 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	1	1	1	3
cPINOT	1	0	0	0

Table 8.4: Comparisons on the Visitor Pattern

is illustrated in Figure A.16). Different from PINOT’s search criteria is that the MUSCAT



Figure 8.8: The MUSCAT Specification on the Visitor Pattern

specification requires both the element and visitor classes to be declared abstract. MUSCAT allows users to mark or unmark a class to be *abstract*. Both cPINOT and PINOT recognize the same Visitor pattern instance implemented in Ant.



## Composite

Table 8.5 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.9 (where its translated C++

	Ant	AWT	JHotDraw	Swing
PINOT	44	3	4	20
cPINOT	33	33	49	118

Table 8.5: Comparisons on the Composite Pattern

code is illustrated in Figure A.5). The specification defines the inheritance and aggre-

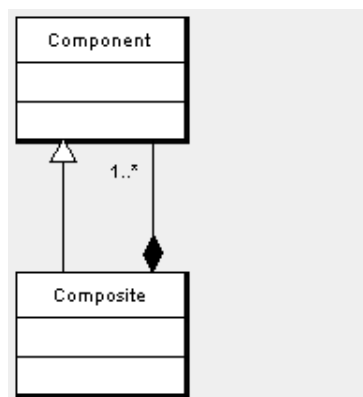


Figure 8.9: The MUSCAT Specification on the Composite Pattern

gation relations between two classes. However, the definition differs from PINOT’s search criteria. First, the class definition does not exclude Component classes that are of type `java.lang.Object`. Further, the inheritance definition is based on the fact that Composite is a sub-type of Component, but it does not reject the case where these two classes are identical. PINOT implements these criteria to prune out such cases. Unfortunately, MUSCAT lacks the notation to specify these criteria. A difference in interpretation is that PINOT accepts an instance of the Composite pattern as long as the Composite and Component derive from a common super type. This explains why cPINOT reports lower number of instances on Ant.

## Chain of Responsibility

Table 8.6 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.10 (where its translated C++

	Ant	AWT	JHotDraw	Swing
PINOT	3	4	5	15
cPINOT	2	3	9	13

Table 8.6: Comparisons on the CoR Pattern

code is illustrated in Figure A.6). The discrepancies in Table 8.6 are due to the differ-

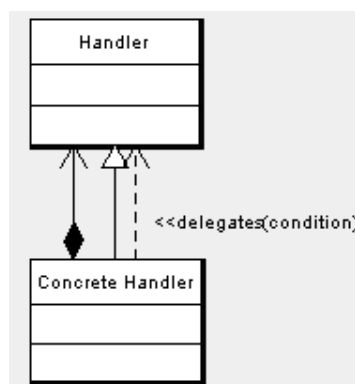


Figure 8.10: The MUSCAT Specification on the CoR Pattern

ence in search criteria: PINOT allows the aggregation relation to be one-to-many. While the MUSCAT specification defines a one-to-one aggregation, it does not specifically reject variable declarations that are arrays (which represent a one-to-many aggregation relation). The reason why cPINOT rejects one-to-many aggregation is only accidental. The MUSCAT specification requires the handler variable (i.e., aggregation relation) to be declared as a field in Concrete Handler and be the only delegating point. If the handler is an array, then the delegating point is usually a local variable and is not captured for the later behavioral analysis.

## Decorator

Table 8.7 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.11 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	4	3	5	15
cPINOT	6	3	7	24

Table 8.7: Comparisons on the Decorator Pattern

is illustrated in Figure A.7). The discrepancies shown in Table 8.7 are due to the reporting

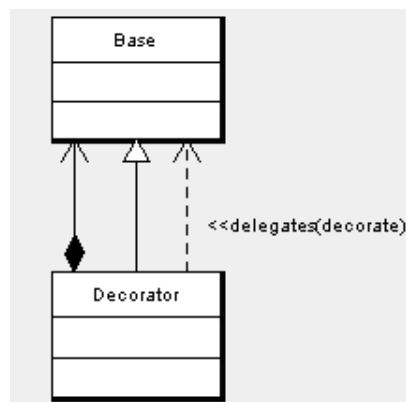


Figure 8.11: The MUSCAT Specification on the CoR Pattern

of pattern instances used by PINOT and cPINOT. PINOT reports per Decorator class, while cPINOT reports per aggregation that satisfies the delegation constraint.

## Flyweight

Table 8.8 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.12 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	35	13	10	54
cPINOT	0	0	0	0

Table 8.8: Comparisons on the Flyweight Pattern

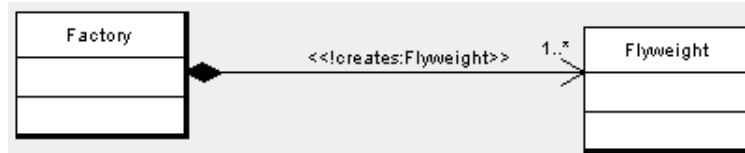


Figure 8.12: The MUSCAT Specification on the Flyweight Pattern

is illustrated in Figure A.9). The specification targets the implementation variant discussed in the GoF book, where a flyweight factory class produces flyweight objects on demand. However, PINOT recognizes other implementation variants, such as including immutable classes and classes that contain static final fields.

### Observer

Table 8.9 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.13 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	5	9	9	68
cPINOT	4	5	2	15

Table 8.9: Comparisons on the Observer Pattern

is illustrated in Figure A.11). PINOT uses the same search criteria as defined in the MUS-



Figure 8.13: The MUSCAT Specification on the Observer Pattern

CAT definition. However, for this particular pattern, PINOT reports the number of methods that implement \*delegates (explained in Table 8.1), while the MUSCAT specification reports a pattern instance whenever one method satisfies the constraint.

## Mediator

Table 8.10 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.14 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	246	170	306	556
cPINOT	107	75	64	192

Table 8.10: Comparisons on the Mediator Pattern

is illustrated in Figure A.10). The MUSCAT specification targets one implementation vari-



Figure 8.14: The MUSCAT Specification on the Mediator Pattern

ant, where the mediator class serves as a communication relay point among a set Colleague objects of the same type. In addition to the the MUSCAT specification, PINOT recognizes pull-model communication and also allows the colleague objects to be different types. Table 8.10 shows that cPINOT recognizes a subset of pattern instances detected by PINOT.

## State

Table 8.11 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.15 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	5	5	3	37
cPINOT	42	33	29	172

Table 8.11: Comparisons on the State Pattern

is illustrated in Figure A.14). PINOT enforces several constraints in addition to the MUSCAT specification: the context class is not declared abstract, anonymous, or have derived

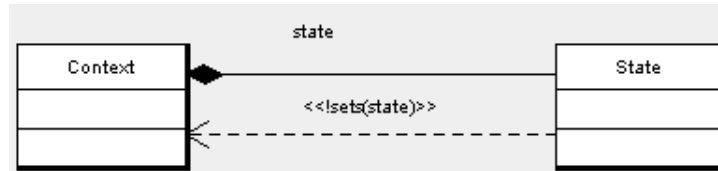


Figure 8.15: The MUSCAT Specification on the State Pattern

classes; the `State` class must be a source class (which is defined in a `.java` file), declared abstract, not an array, and not derived from the same root class as `Context`). `cPINOT` is able to detect the pattern instances (as explained in Table 8.11) identified by `PINOT`. The remaining instances are caused by the looser criteria on both the `Context` and `state` classes. `MUSCAT` can define most of these extra constraints, but not all. For class constraints, users can specify a class to be abstract by checking the *abstract* modifier checkbox or specify that a class has no derived classes by checking the *leaf* modifier. However, `MUSCAT` does not allow specifying if a class is anonymous or is a source class. Such constraints can be easily introduced to the `MUSCAT` language as stereotypes for classes. For inter-class relations, `MUSCAT` users can use `<<independent>>` (explained in Table 8.1) to specify that two classes do not derive from the same root class. However, `MUSCAT` lacks the notation to define non-existence of certain relations, which is useful in this case to specify that there is no one-to-many relation between `Context` and `State`.

### Strategy

Table 8.12 compares the number of pattern instances reported by `cPINOT` and `PINOT`. `cPINOT` used the `MUSCAT` definition shown in Figure 8.16 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	19	54	51	96
cPINOT	105	154	199	393

Table 8.12: Comparisons on the Strategy Pattern

is illustrated in Figure A.15). `PINOT` enforces the following constraints in addition to the

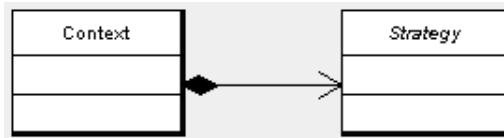


Figure 8.16: The MUSCAT Specification on the Strategy Pattern

MUSCAT specification: the context class is not declared abstract, anonymous, or have derived classes; the Strategy class must be a source class and is not derived from the same root class as context). cPINOT detects the pattern instances (as illustrated in Table 8.12) identified by PINOT. The remaining instances are caused by the looser criteria defined in the MUSCAT specification.

## Bridge

Table 8.13 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.17 (where its translated C++

	Ant	AWT	JHotDraw	Swing
PINOT	5	15	107	142
cPINOT	1	14	4	13

Table 8.13: Comparisons on the Bridge Pattern

code is illustrated in Figure A.4). The MUSCAT specification requires both interface and

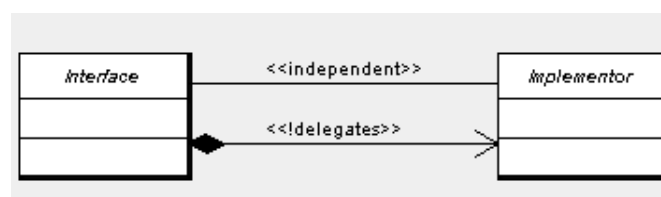


Figure 8.17: The MUSCAT Specification on the Bridge Pattern

implementor to be declared abstract, while PINOT only requires interface to be declared abstract. The difference in search criteria is reflected in terms of the number of pattern instances shown in Table 8.13.

## Proxy

Table 8.14 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.18 (where its translated C++ code

	Ant	AWT	JHotDraw	Swing
PINOT	27	13	107	142
cPINOT	55	50	63	175

Table 8.14: Comparisons on the Proxy Pattern

is illustrated in Figure A.12). The MUSCAT specification defines inheritance constraints

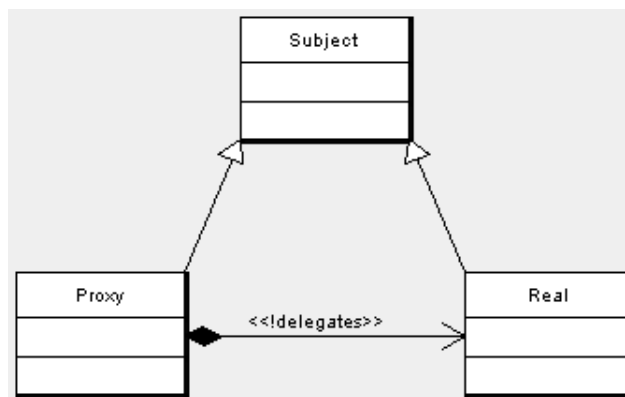


Figure 8.18: The MUSCAT Specification on the Proxy Pattern

over the Proxy and Real classes and the exclusive one-way communication. In addition to the specification, PINOT rejects anonymous classes and restricts Proxy and Real to be different classes. These search criteria are essential based on the GoF definition of the pattern, however MUSCAT lacks language constructs to specify these constraints. In Table 8.14, PINOT reports per identified Proxy class, while cPINOT reports per identified aggregation relation (i.e., for each variable in Proxy that satisfies the pattern constraint).

## Adapter

Table 8.15 compares the number of pattern instances reported by cPINOT and PINOT. cPINOT used the MUSCAT definition shown in Figure 8.19 (where its translated C++ code



	Ant	AWT	JHotDraw	Swing
PINOT	13	15	107	142
cPINOT	496	14	180	381

Table 8.15: Comparisons on the Adapter Pattern

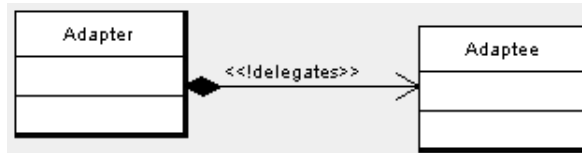


Figure 8.19: The MUSCAT Specification on the Adapter Pattern

is illustrated in Figure A.3). The MUSCAT specification only targets the exclusive one-way communication between the two classes `Adapter` and `Adaptee`. The Proxy and Adapter patterns are very similar, where both require the same communication style, except for the inheritance and other class property constraints. In addition to the MUSCAT specification, PINOT requires both `Adapter` and `Adaptee` (1) to be concrete classes (i.e., not declared abstract or as interfaces) that (2) are not derived from the same root class. Further, PINOT ensures that there (3) exists actual object creation for the adaptee. That is, the `Adaptee` class is instantiated by `Adapter`. MUSCAT does provide language constructs to describe these constraints. The first two constraints involve property negation. MUSCAT can specify the existence of a constraint, but not the absence. The third constraint can be added to MUSCAT as a stereotype over an aggregation relation.

### Timing Results

Figure 8.20 shows the timing results running PINOT and cPINOT on the same set of applications: Ant, AWT, JHotDraw, and Swing. Note that the timing results for PINOT are based on Figure 5.1, reported previously in Section 5.2, and both timing results for PINOT and cPINOT shown in the Figure 8.20 include the entire analysis and I/O times. The timing differences are caused by their differences in the internal search and reporting mechanisms. PINOT internally adopts several speed optimization techniques, thus it is generally faster

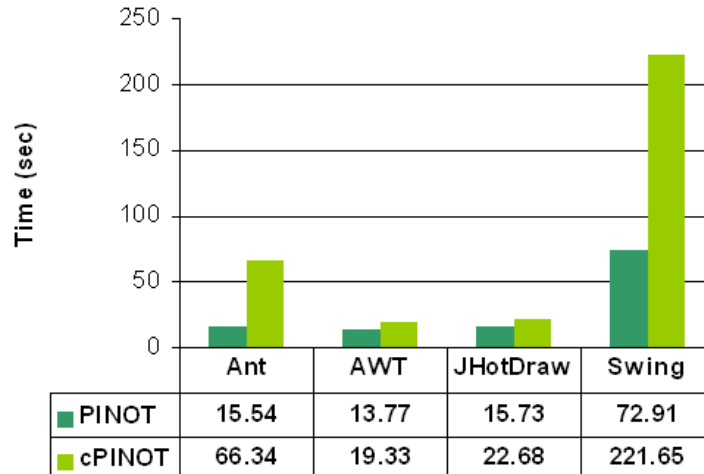


Figure 8.20: Timing Results for PINOT vs. cPINOT

than cPINOT. The optimizations include the way how each pattern is analyzed and how they are run together. Depending on the property (i.e., class, method, or field property) of a pattern, the starting point (i.e., class, method, or field declaration) of the search can affect speed. For example, searching from each method/field declaration is more expensive than searching from each class, since in an application there are generally more methods/fields than classes. cPINOT simply analyzes an application based on the input specifications. Users do not have control to the internal search mechanism. Another optimization is that PINOT analyzes similar patterns together, whereas cPINOT treats each input specification independently and processes them one at a time in a sequential order. An issue that may cause different timing results is that PINOT can filter irrelevant types (such as `java.lang.Object`, or other primitive types) during the search, while cPINOT has no such control. Finally, in terms of reporting, cPINOT is more verbose than PINOT. PINOT only reports when a pattern instance is found, while cPINOT reports whenever a MUSCAT notation is found (e.g., a one-to-many association). Therefore, cPINOT spends more I/O processing time.

To further investigate the bottleneck in the search mechanism of cPINOT, we measure the time cPINOT takes to run each specification. Figures 8.21, 8.22, and 8.23 show the timing results for each specification. Here, we measure only the analysis times for each

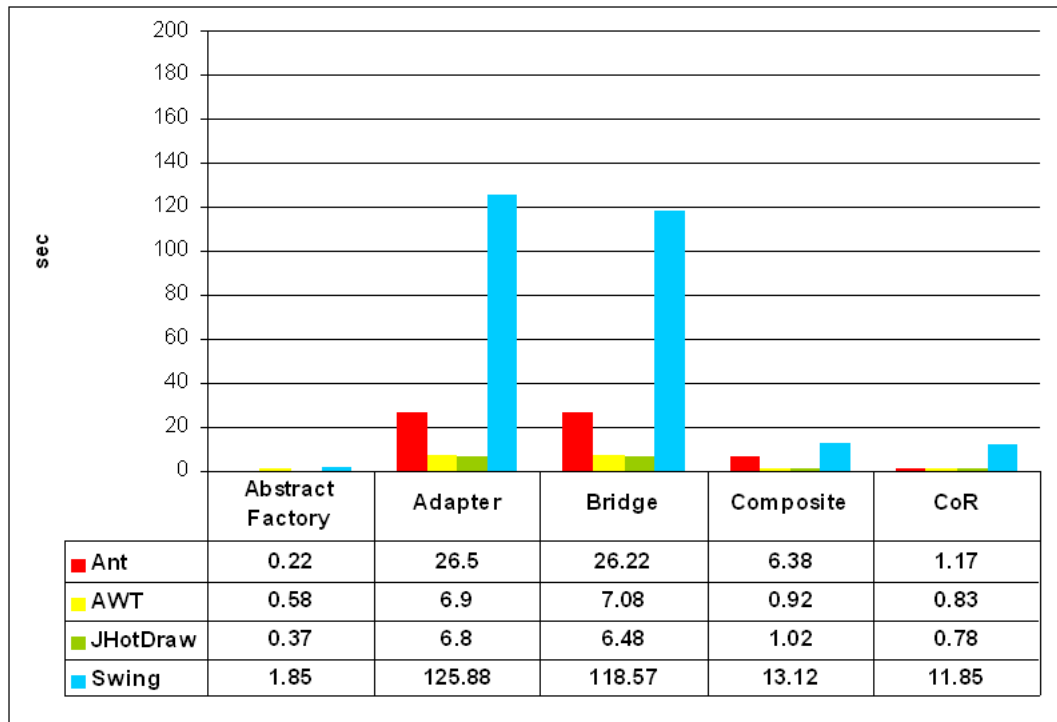


Figure 8.21: Timing Results for cPINOT per Pattern (Part I)

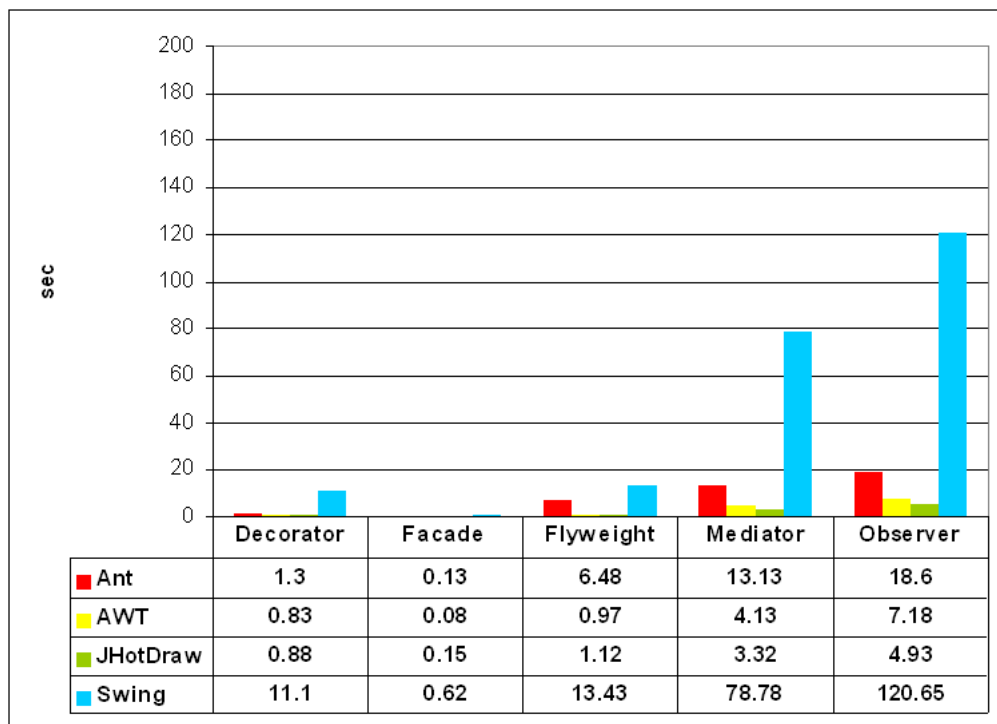


Figure 8.22: Timing Results for cPINOT per Pattern (Part II)

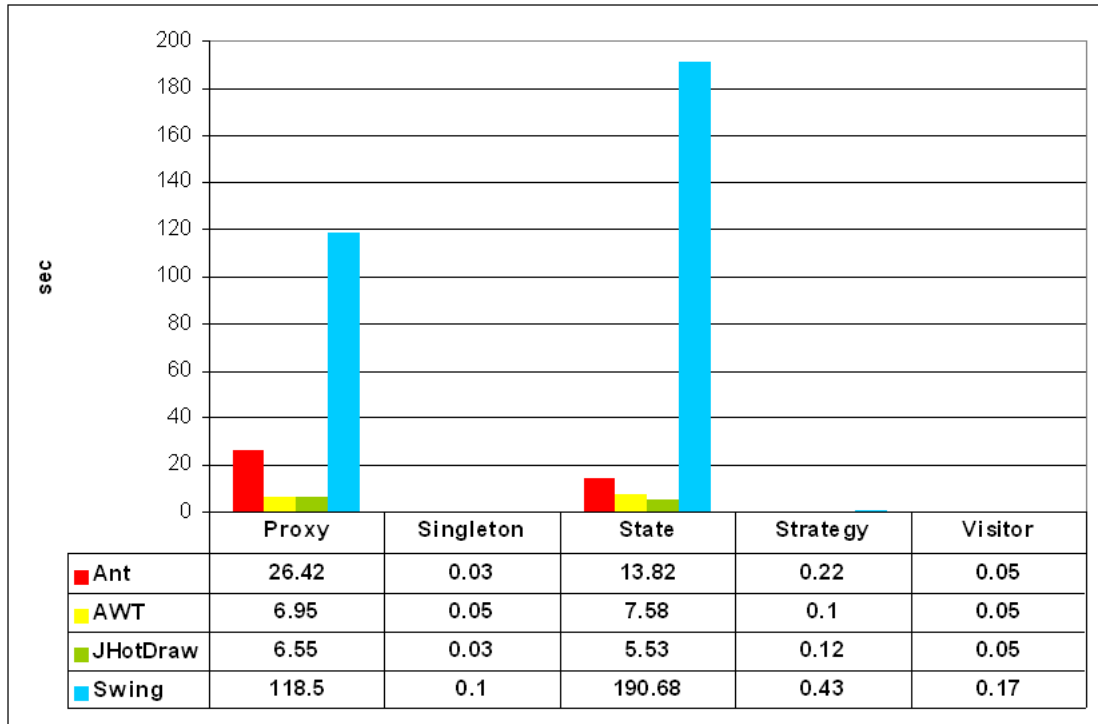


Figure 8.23: Timing Results for cPINOT per Pattern (Part III)

pattern. The patterns that take relatively longer than rest are the ones require analyzing delegation. A delegation is based on some field or variable declaration. This requires cPINOT to iterate over each field declaration in a class and then verify specific delegation behavior. The State pattern is relatively more complicated than other patterns, since its delegation requires additional checking for the setting of attributes.

Among the four applications, Ant, AWT, and JHotDraw, are similar in size (in terms of the number of classes, as shown in Figure 5.1). Yet, Ant seems to take much longer time for cPINOT to analyze than for the other two applications. While Ant is not significantly larger in size, the application itself is relatively more complex. That is, the source code declares more methods and fields than the other applications. The fact that cPINOT tends to spend more time for such applications is a drawback in MUSCAT. MUSCAT lacks language constructs for users to specify whether to halt or continue a search analysis process whenever a pattern instance is identified with a class. In many cases, we are only interested in knowing

whether a class participates in a certain pattern without the need to identify all kinds of participating combinations with other participants (which can be classes or methods). PINOT optimizes this process by caching searched results, so it can later skip searched participants for certain patterns and reporting preferences.

In general, cPINOT takes longer time than PINOT to analyze patterns. There is a trade-off between flexibility versus speed. PINOT is faster, but it is not customizable. However, we consider the extra time to be reasonable in practice. The Swing application is relatively more time-consuming than the rest, but it is still able to finish processing 15 specifications in less than 3 minutes. We expect that users will not detect patterns in their source code as frequently as they perform other activities, such as compiling their code.

### 8.5.2 Beyond GoF

To explore MUSCAT's language capability, we try to define three commonly be used patterns that are beyond the GoF book. Figure 8.24 shows an example of a Factory pattern, that is similar to the Abstract Factory pattern except both the Factory and Product classes can be declared non-*abstract*. Figure B.1 illustrates C++ code translated from this MUSCAT specification. This design pattern is recognized as the Factory Method pattern by PINOT [20], FUJABA [6], and HEDGEHOG [27]. Table 8.16 shows the results in terms of the number of factory methods found.

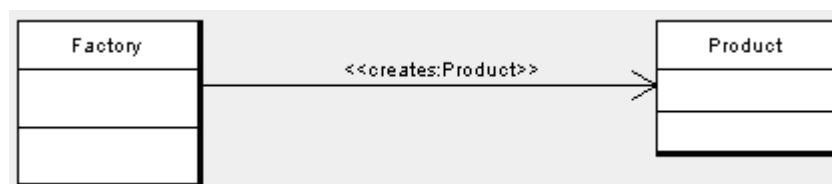


Figure 8.24: Variant of Abstract Factory Pattern

Figure 8.25 shows a variant of the previous example that requires Product class to be a singleton (Reference [9] describes the implementation and design of such design pattern). Figure B.2 illustrates C++ code translated from this MUSCAT specifications. Table 8.17

	Ant	AWT	JHotDraw	Swing
cPINOT	36	207	198	244

Table 8.16: Results for the Variant of Abstract Factory Pattern

shows the results in terms of pattern instances found. Although this pattern appears to

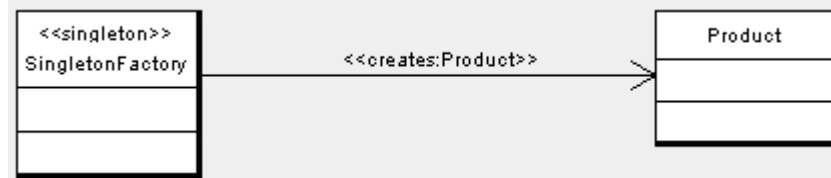


Figure 8.25: Variant of Abstract Factory Pattern with Singleton Factory

	Ant	AWT	JHotDraw	Swing
cPINOT	0	0	0	0

Table 8.17: Results for the Variant of Abstract Factory Pattern with Singleton Factory

be absent in our benchmark applications, we have tested its correctness in a smaller test application.

Figure 8.26 shows a variant of CoR that requires the aggregation relation to be one-to-many. Figure B.3 illustrates the C++ code translated from this MUSCAT specification. The GoF book [38] describes such implement variant as a combination of the Composite and the CoR patterns. Table 8.18 shows the results in terms of pattern instances found.

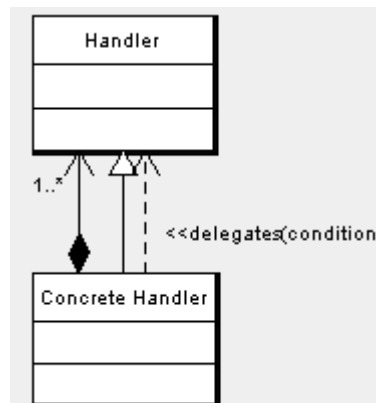


Figure 8.26: One-to-many CoR

	Ant	AWT	JHotDraw	Swing
cPINOT	1	3	9	12

Table 8.18: Results for the One-to-many CoR Pattern

## 8.6 Discussion

The examples listed in Section 8.5.2 are only slight variations of the GoF patterns. MUSCAT is still limited for describing more complicated patterns. For example, *viognier* cannot translate a specification that contains more than one aggregation association. The reasons why this is not yet allowed in *viognier* are due to usability and performance. For any given MUSCAT specification, *viognier* determines the starting point (whether to iterate through the classes, methods, or fields) for pattern detection. Certain MUSCAT notation maps to certain starting point (for more details, see Section 8.4). Let us refer to such notations as *starter* notations. If one or more starter notations are present in a specification, then *viognier* cannot determine the right pattern detection starting point. However, such specifications are not necessarily valid. For example, a specification with only two class notations is an invalid specification. In the future, *viognier* will need to be able to report invalid specifications. For valid specifications, *viognier* can, perhaps, apply multiple nested iterations. However, this requires sophisticated design to avoid  $O(n^n)$  time complexity. In the future, we want to allow users to specify search priority for each starter notation. That is, users might be allowed to annotate their specifications with integers that designate the order in which such notations will be analyzed.

MUSCAT can be implemented with pattern detection APIs other than the PINOT API, but such an implementation will require the API to have the ability to perform various behavioral analyses. As an example, one can use the SPINE language to implement the structural-driven patterns specified in MUSCAT, but not all of the behavior-driven patterns. This is because MUSCAT expresses patterns at the design level, while SPINE expresses patterns at the code level. That is, MUSCAT is more expressive in terms of inter-class rela-

tionships, while SPINE is more expressive in terms of class, method, and field declarations. For example, MUSCAT can express the flyweight pattern (defined in Figure 8.12) based on the GoF definition, but this particular behavioral constraint cannot be expressed in SPINE.



## Chapter 9

# Conclusion and Future Work

In the process of software development, developers make use of various tools and techniques to help them understand and eventually fine tune their software projects. Program understanding tools today fall short in extracting the architectural design of a piece of software. Since the mid 90s, when the GoF book [38] was published, design patterns have shaped the way programmers program in object-oriented languages. The idea of design patterns is to ensure extensibility and maintainability in object-oriented design. Today, design patterns have become widely used in practice. Each design pattern is associated with an architectural design intent and a few coding guidelines. We believe by extracting design patterns from source code, we can bring program understanding to the design level.

We began our research by studying existing pattern recognition tools. In Section 2.1, we compared these tools and analyzed their approaches. We discovered that the accuracy of a pattern recognition tool is relative to its interpretation of a design pattern. In Section 2.2, we discussed the relationship between pattern interpretation and implementation variants and their effect on pattern recognition. We found that most of these tools are able to extract the structural aspect of a design pattern, but they cannot effectively capture the behavioral aspect, that is the program intent. We illustrated some of these examples in Section 2.3.

We built our own pattern recognition tool, PINOT, based on solely static analysis tech-

niques. Our goal was to recognize all design patterns illustrated in the GoF book. However, not all of the patterns are detectable. Chapter 3 illustrated our reclassification of the GoF patterns, and Chapter 4 explained how we built PINOT based on our theory. Chapter 5 presented the implementation of PINOT, as well as the timing and accuracy results tested on several benchmark applications. Our results proved PINOT to be faster and more accurate than existing pattern recognition tools.

While the GoF patterns are widely used in practice, there are many other useful design patterns that are beyond the GoF book. Some of these design patterns are variations of the GoF patterns and some are targeted to specific software domains. We felt a need to further extend the recognition capability of PINOT by allowing users to specify and analyze their own design patterns. We began this part of the research by studying why and how design patterns are formalized. Chapter 6 discussed various purposes (including software modeling, code refactoring and generation, and pattern detection) and techniques to formalize design patterns. Chapter 7 compared languages defined for pattern detection. We presented the design and implementation of our pattern detection language, MUSCAT, in Chapter 8. In addition, we defined a set of PINOT API to facilitate the implementation of MUSCAT. Finally, we constructed a pattern recognition tool using MUSCAT to define the detectable GoF patterns and compared it against PINOT. Our results, while promising, showed PINOT to be faster. We then discussed the tradeoff between effectiveness versus flexibility.

In the future, we want to continue this research in two directions: enhancing PINOT and refining MUSCAT. Section 5.3 discussed the future work for PINOT: upgrading PINOT to recognize the latest version of Java extending PINOT's recognition capability; and integrating PINOT with other software development tools. Section 8.6 discussed the future work for MUSCAT: fine-tuning the performance aspects of the implementation; providing a better error reporting mechanism when interpreting a MUSCAT specification; and introducing new language constructs to increase usability.

# Appendix A

## Viognier Code Generation

### – the GoF Patterns

*Viognier* is a Python script that translates a MUSCAT specification (in XMI format) into C++ code that uses the PINOT API. This appendix illustrates the C++ code generated from each MUSCAT specification shown in Section 8.5.1. Every complete C++ code starts with a common header code. Due to length, we present the header code once in Figure A.1. Figures A.2 to A.16 each shows the detection function translated from the corresponding MUSCAT specification.

```

#include "ast.h"
#include "control.h"
#include "symbol.h"
#include <vector>
#include <iostream>

using namespace std;

#define cout cout
#define Coutput cout

extern "C"
{
    int getCount();
    void FindPattern0(SymbolTables*);
}

extern bool isAbstract
    (TypeSymbol*);
extern bool isSingletonClass
    (SymbolTables*, TypeSymbol*);
extern bool isFacadeClass
    (TypeSymbol*);
extern bool isFactoryMethod
    (SymbolTables*, MethodSymbol*);
extern bool createsFlyweights
    (VariableSymbol*, TypeSymbol*);
extern VariableSymbol *isVisitorMethod
    (MethodSymbol*);
extern bool *bang_sets
    (SymbolTables*, TypeSymbol *setter, VariableSymbol *vsym);
extern bool star_delegates
    (SymbolTables*, AstVariableDeclarator*, TypeSymbol*,
    TypeSymbol*);
extern bool bang_delegates
    (SymbolTables *, TypeSymbol*, TypeSymbol*);
extern bool cond_delegates
    (SymbolTables*, VariableSymbol*);
extern bool decor_delegates
    (SymbolTables *, VariableSymbol*);
extern bool areIndependentHierarchies
    (SymbolTables*, TypeSymbol*, TypeSymbol*);

int getCount() { return 1; }

```

Figure A.1: The Generated C++ Code that Precedes Every Detection Code

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table = sym_tables->getDelegationTable();

    for (unsigned m = 0; m < ms_table -> size(); m++)
    {
        MethodSymbol *method = (*ms_table)[m];
        if ( isFactoryMethod(sym_tables, method) )
        {
            Coutput << "abstractfactory|<<creates:AbstractProduct>>|"
                << method->Utf8Name ()
                << "|"
                << method->containing_type->file_symbol->FileName ()
                << endl;
            TypeSymbol *leftEndType = method -> containing_type;
            if ( isAbstract( leftEndType ) )
            {
                Coutput << "abstractfactory|AbstractFactory|"
                    << leftEndType->Utf8Name ()
                    << leftEndType->file_symbol->FileName ()
                    << endl;
            }
            TypeSymbol *rightEndType = method -> Type();
            if ( isAbstract( rightEndType ) )
            {
                Coutput << "abstractfactory|AbstractProduct|"
                    << rightEndType->Utf8Name ()
                    << leftEndType->file_symbol->FileName ()
                    << endl;
            }
        }
    }
}

```

Figure A.2: The Generated C++ Code that Detects the Abstract Factory Pattern (as defined in Figure 8.7)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table = sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables();
            i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration-> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators();
                vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "adapter|one-to-one|"
                        << vd->symbol->Utf8Name ()
                        << "|"
                        << vd->symbol->ContainingType ()
                            ->file_symbol->FileName ()
                        << endl;
                    if (bang_delegates(sym_tables, type, containing_type))
                    {
                        Coutput << "adapter|Adapter|"
                            << type->Utf8Name ()
                            << "|"
                            << type->file_symbol->FileName ()
                            << endl;
                        Coutput << "adapter|Adaptee|"
                            << containing_type->Utf8Name ()
                            << "|"
                            << containing_type->file_symbol->FileName ()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.3: The Generated C++ Code that Detects the Adapter Pattern (as defined in Figure 8.19)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table = sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables();
            i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators();
                vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "bridge|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if (bang_delegates(sym_tables,
                                        type, containing_type) &&
                        isAbstract(type) &&
                        isAbstract(containing_type) &&
                        areIndependentHierarchies(sym_tables, type,
                                                  containing_type))
                    {
                        Coutput << "bridge|Interface|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "bridge|Implementor|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.4: The Generated C++ Code that Detects the Bridge Pattern (as defined in Figure 8.17)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type =
                    type -> IsOnetoMany(vd->symbol, d_table);
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "composite|one-to-many|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if ((type -> IsSubtype(containing_type)))
                    {
                        Coutput << "composite|Composite|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName() << endl;
                        Coutput << "composite|Component|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.5: The Generated C++ Code that Detects the Composite Pattern (as defined in Figure 8.9)



```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "cor|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if ((type -> IsSubtype(containing_type)) &&
                        (cond_delegates(sym_tables, vd->symbol)))
                    {
                        Coutput << "cor|Concrete Handler|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "cor|Handler|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.6: The Generated C++ Code that Detects the CoR Pattern (as defined in Figure 8.10)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "decorator|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if ((type -> IsSubtype(containing_type)) &&
                        (decor_delegates(sym_tables, vd->symbol)))
                    {
                        Coutput << "decorator|Decorator|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "decorator|Base|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.7: The Generated C++ Code that Detects the Decorator Pattern (as defined in Figure 8.11)

```
void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        if (isFacadeClass(type))
        {
            Coutput << "facade|<<facade>>|"
                << type->Utf8Name ()
                << "|"
                << type->file_symbol->FileName ()
                << endl;
        }
    }
}
```

Figure A.8: The Generated C++ Code that Detects the Facade Pattern (as defined in Figure 8.6)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type =
                    type -> IsOnetoMany(vd->symbol, d_table);
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "flyweight|one-to-many|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if (createsFlyweights(vd->symbol, containing_type))
                    {
                        Coutput << "flyweight|Factory|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "flyweight|Flyweight|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.9: The Generated C++ Code that Detects the Flyweight Pattern (as defined in Figure 8.12)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type =
                    type -> IsOnetoMany(vd->symbol, d_table);
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "mediator|one-to-many|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if (bang_delegates(sym_tables, type, containing_type))
                    {
                        Coutput << "mediator|Mediator|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName() << endl;
                        Coutput << "mediator|Colleague|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.10: The Generated C++ Code that Detects the Mediator Pattern (as defined in Figure 8.14)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type =
                    type -> IsOnetoMany(vd->symbol, d_table);
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "observer|one-to-many|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName() << endl;
                    if ((star_delegates(sym_tables,
                                        vd, type, containing_type)))
                    {
                        Coutput << "observer|Subject|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName() << endl;
                        Coutput << "observer|Observer|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.11: The Generated C++ Code that Detects the Observer Pattern (as defined in Figure 8.13)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "proxy|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName() << endl;
                    if (bang_delegates(sym_tables,
                        type, containing_type) &&
                        (type -> IsFamily(containing_type)))
                    {
                        Coutput << "proxy|Proxy|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName() << endl;
                        Coutput << "proxy|Real|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.12: The Generated C++ Code that Detects the Proxy Pattern (as defined in Figure 8.18)

```
void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        if (isSingletonClass(sym_tables, type))
        {
            Coutput << "singleton|<<singleton>>|"
                << type->Utf8Name ()
                << "|"
                << type->file_symbol->FileName ()
                << endl;
        }
    }
}
```

Figure A.13: The Generated C++ Code that Detects the Singleton Pattern (as defined in Figure 8.5)



```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "state|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if ((bang_sets(sym_tables,
                        containing_type, vd->symbol)))
                    {
                        Coutput << "state|Context|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "state|State|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.14: The Generated C++ Code that Detects the State Pattern (as defined in Figure 8.15)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type = vd -> symbol -> Type();
                if (containing_type && containing_type -> file_symbol)
                {
                    Coutput << "strategy|one-to-one|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if (isAbstract(containing_type))
                    {
                        Coutput << "strategy|Context|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName()
                            << endl;
                        Coutput << "strategy|Strategy|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure A.15: The Generated C++ Code that Detects the Strategy Pattern (as defined in Figure 8.16)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned m = 0; m < ms_table -> size(); m++)
    {
        MethodSymbol *method = (*ms_table)[m];
        if ( VariableSymbol *param = isVisitorMethod(method) )
        {
            Coutput << "visitor|<<accepts (Visitor)>>|"
                << method->Utf8Name () << "|"
                << method->containing_type->file_symbol->FileName ()
                << endl;
            TypeSymbol *leftEndType = method -> containing_type;
            if ( isAbstract( leftEndType ) )
            {
                Coutput << "visitor|Element|"
                    << leftEndType->Utf8Name ()
                    << leftEndType->file_symbol->FileName ()
                    << endl;
            }
            TypeSymbol *rightEndType = param -> Type();
            if ( isAbstract( rightEndType ) )
            {
                Coutput << "visitor|Visitor|"
                    << rightEndType->Utf8Name ()
                    << leftEndType->file_symbol->FileName ()
                    << endl;
            }
        }
    }
}

```

Figure A.16: The Generated C++ Code that Detects the Visitor Pattern (as defined in Figure 8.8)

## Appendix B

### Viognier Code Generation

#### – Patterns Beyond GoF

This appendix lists the C++ code translated from the MUSCAT specifications discussed in Section 8.5.2. Figures B.1 to B.3 each shows the detection function translated from the corresponding MUSCAT specification, while using the same header code shown in Figure A.1.

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned m = 0; m < ms_table -> size(); m++)
    {
        MethodSymbol *method = (*ms_table)[m];
        if ( isFactoryMethod(sym_tables, method) )
        {
            Coutput << "example0|
                << creates:Product>>|"
                << method->Utf8Name ()
                << "| "
                << method->containing_type
                    ->file_symbol->FileName ()
                << endl;
        }
    }
}

```

Figure B.1: The Generated C++ Code that Detects a Variant of the Abstract Factory Pattern Pattern (as defined in Figure 8.24)

```

void FindPattern0 (SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned m = 0; m < ms_table -> size(); m++)
    {
        MethodSymbol *method = (*ms_table)[m];
        if ( isFactoryMethod(sym_tables, method) )
        {
            Coutput << "example1|
                << creates:Product>>|"
                << method->Utf8Name() << "|"
                << method->containing_type
                    ->file_symbol->FileName()
                << endl;
            TypeSymbol *leftEndType =
                method -> containing_type;
            if ( isSingletonClass(sym_tables, leftEndType) )
            {
                Coutput << "example1|SingletonFactory|"
                    << leftEndType->Utf8Name()
                    << leftEndType->file_symbol->FileName()
                    << endl;
            }
        }
    }
}

```

Figure B.2: The Generated C++ Code that Detects a Variant of the Abstract Factory Pattern with Singleton Factory (as defined in Figure 8.25)

```

void FindPattern0(SymbolTables *sym_tables)
{
    ClassSymbolTable *cs_table =
        sym_tables->getClassSymbolTable();
    MethodSymbolTable *ms_table =
        sym_tables->getMethodSymbolTable();
    DelegationTable *d_table =
        sym_tables->getDelegationTable();

    for (unsigned c = 0; c < cs_table -> size(); c++)
    {
        TypeSymbol *type = (*cs_table)[c];
        for (unsigned i = 0;
            i < type -> declaration-> NumInstanceVariables(); i++)
        {
            AstFieldDeclaration* field_decl =
                type -> declaration -> InstanceVariable(i);
            for (unsigned vi = 0;
                vi < field_decl -> NumVariableDeclarators(); vi++)
            {
                AstVariableDeclarator* vd =
                    field_decl -> VariableDeclarator(vi);
                TypeSymbol *containing_type =
                    type -> IsOnetoMany(vd->symbol, d_table);
                if (containing_type &&
                    containing_type -> file_symbol)
                {
                    Coutput << "example2|one-to-many|"
                        << vd->symbol->Utf8Name() << "|"
                        << vd->symbol->ContainingType()
                            ->file_symbol->FileName()
                        << endl;
                    if ((type -> IsSubtype(containing_type)) &&
                        (cond_delegates(sym_tables, vd->symbol)))
                    {
                        Coutput << "example2|Concrete Handler|"
                            << type->Utf8Name() << "|"
                            << type->file_symbol->FileName() << endl;
                        Coutput << "example2|Handler|"
                            << containing_type->Utf8Name() << "|"
                            << containing_type->file_symbol->FileName()
                            << endl;
                    }
                }
            }
        }
    }
}

```

Figure B.3: The Generated C++ Code that Detects a One-to-many CoR (as defined in Figure 8.26)

# Bibliography

- [1] ArgoUML. <http://argouml.tigris.org/>.
- [2] aspectj. <http://www.eclipse.org/aspectj/>.
- [3] data & object factory: DEVELOPER TRAINING. <http://www.dofactory.com/Patterns/Patterns.aspx>.
- [4] Eclipse. <http://www.eclipse.org/>.
- [5] FluffyCat.com: Java Design Patterns Reference and Examples. <http://www.fluffycat.com/java-design-patterns/>.
- [6] FUJABA. <http://www.fujaba.de>.
- [7] Improved Observer/Observable with Remote Capability. <http://www2.sys-con.com/ITSG/virtualcd/java/archives/0210/schwell/index.html>.
- [8] Java Resources: Design Patterns. <http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/>.
- [9] Java Tips: The Singleton Pattern. <http://www.java-tips.org/java-se-tips/java.lang/the-singleton-pattern.html>.
- [10] JHotDraw. <http://www.jhotdraw.org>.
- [11] Jikes. <http://jikes.sourceforge.net/>.
- [12] JSR 199: Java™ Compiler API. <http://jcp.org/en/jsr/detail?id=199>.
- [13] LePUS2. <http://www.eden-study.org/lepus/>.
- [14] MDA. <http://www.omg.org/mda/>.
- [15] Oracle JDeveloper. <http://www.oracle.com/technology/products/jdev/index.html>.
- [16] PatHPC: Workshop on Patterns in High Performance Computing. <http://charm.cs.uiuc.edu/patHPC>.



- [17] Pattern Box Eclipse Tool. <http://www.patternbox.com>.
- [18] Pattern Stories: JavaAWT. <http://wiki.cs.uiuc.edu/PatternStories/JavaAWT>.
- [19] PLoP: Pattern Languages of Programs. <http://hillside.net/conferences/plop.htm>.
- [20] The PINOT Website. <http://www.cs.ucdavis.edu/~shini/research/pinot>.
- [21] UNIMOD. <http://www.unimod.org>.
- [22] Herve Albin-Amiot, Pierre Cointe, Yann-Gael Guehéneuc, and Narendra Jussien. Instantiating and detecting design patterns: putting bits and pieces together. In *ASE '01: Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, Nov. 2001.
- [23] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proc. of the 6th International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.
- [24] Angel Asencio, Sam Cardman, David Harris, and Ellen Laderman. Relating expectations to automatically recovered design patterns. In *WCRE*, pages 87–96, 2002.
- [25] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from C++ source code. In *Proc. of the International Conference on Software Maintenance*, pages 305–314. IEEE Computer Society Press, September 2003.
- [26] Jagdish Bansiya. Automating design-pattern identification – DP++ is a tool for C++ programs. *Dr. Dobbs Journal*, 1998.
- [27] Alex Blewitt. *HEDGEHOG: Automatic Verification of Design Patterns in Java*. PhD thesis, University of Edinburgh, 2005.
- [28] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in Java. In *The 20th International Conference on Automated Software Engineering*, pages 224–232, 2005.
- [29] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.
- [30] Kyle Brown. Design Reverse Engineering and Automated Design Pattern Detection in SmallTalk. Master’s thesis, North Carolina State University, 1998.
- [31] William H. Brown, Raphael Malveau, Hays W. McCormick, and Thomas J. Mowbray. *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, New York, 1998.

- [32] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, 1996.
- [33] A. H. Eden, A. Yehudai, and J. Gil. Precise specification and automatic application of design patterns. In *ASE '97: Proceedings of the 12th International Conference on Automated Software Engineering (formerly: KBSE)*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.
- [34] Johan Fabry and Tom Mens. Language independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures*, February 2004.
- [35] Rudolf Ferenc, Árpád Beszédes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *The 21st International Conference on Software Maintenance*, pages 295–304, 2005.
- [36] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing design patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybern.*, 15(4):669–682, 2002.
- [37] Erich Gamma. Becoming a Programming Picasso with JHotDraw. <http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jhotdraw.html>.
- [38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
- [39] Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. *ACM SIGPLAN Notices*, 40(10):97–116, 2005.
- [40] Yann-Gael Guehéneuc and Narendra Jussien. Using explanations for design patterns identification. In *Proceedings of the 1st IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, August 2001.
- [41] Yann-Gael Guehéneuc, Houari Shraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 172–181, Nov 2004.
- [42] Yann-Gal Gueheneuc and Herv Albin-Amiot. Recovering binary class relationships: putting icing on the UML cake. In John Vlissides and Doug C. Schmidt, editors, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2004)*, pages 301–314, Vancouver, Canada, 2004. ACM Press.
- [43] Alain Le Guennec, Gerson Sunye, and Jean Marc Jezequel. Precise modeling of design patterns. *Lecture Notes in Computer Science*, pages 482–497, 2003.

- [44] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system-specific, static analyses. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82, 2002.
- [45] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 161–173, New York, NY, USA, 2002. ACM Press.
- [46] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *Proc. of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103. IEEE Computer Society Press, May 2003.
- [47] Allen Holub. *Holub on Patterns: Learning Design Patterns by Looking at Code*. APress, 2004.
- [48] Rudolf Keller, Reinhard Shauer, Sebastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proc. of the 21st International Conference on Software Engineering*, pages 226–235. IEEE Computer Society Press, May 1999.
- [49] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.
- [50] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. Micro pattern evolution. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 40–46, New York, NY, USA, 2006. ACM Press.
- [51] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise modeling of design patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.
- [52] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11. Australian Computer Society, Inc., 2002.
- [53] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, Reading, Massachusetts, 2005.
- [54] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacobson.
- [55] Jorg Niere, Matthias Meyer, and Lothar Wendehals. User-driven adaption in rule-based pattern recognition. Technical report, University of Paderborn, Paderborn, Germany, 2004.

- [56] Jorg Niere, Wilhelm Shafer, Jorg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE*, pages 338–348. IEEE Computer Society Press, May 2002.
- [57] Jorg Niere, Jorg P. Wadsack, and Lothar Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the 11th IEEE International Workshop on Program Comprehension*, pages 274–279. IEEE Computer Society Press, May 2003.
- [58] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. Software Metrics by Architectural Pattern Mining. In *Proceedings of the International Conference on Software: Theory and Practice*, pages 325–332. 16th IFIP World Computer Congress, August 2000.
- [59] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. An approach for reverse engineering of design patterns. *Software Systems Modeling*, pages 55–70, 2005.
- [60] Jochen Seemann and Jrgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, 1998.
- [61] Nija Shi and Ronald A. Olsson. Reverse engineering design patterns from Java source code. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18–22 Spetember, Tokyo, Japan*, pages 123–134. IEEE Society, 2006.
- [62] Jason McC. Smith and Davis Stotts. SPQR: flexible automated design pattern extraction from source code. In *ASE '03: The 18th International Conference on Automated Software Engineering*, pages 215–224. IEEE Computer Society Press, October 2003.
- [63] Stephen Stelting and Olav Maassen. *Applied Java Patterns*. Prentice Hall, Palo Alto, California, 2002.
- [64] Nikolaos Tsantalis and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006.
- [65] Marek Vokáč. An efficient tool for recovering design patterns from C++ code. *Journal of Object Technology*, 5(2), March-April 2006.
- [66] Lothar Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE Workshop on Dynamic Analysis (WODA)*, pages 29–32. IEEE Computer Society Press, May 2003.
- [67] Lothar Wendehals. Specifying patterns for dynamic pattern instance recognition with UML 2.0 sequence diagrams. In *Proc. of the 6<sup>th</sup> Workshop Software Reengineering (WSR)*, volume 24/2, pages 63–64, May 2004.