Feature-Oriented Design Pattern Detection in OO Systems

Feature-Oriented Design Pattern Detection in Object-Oriented Systems

By

Lei Hu, B.Eng.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

M.A.Sc.

McMaster University

MASTER OF APPLIED SCIENCE (2007) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE: Feature-Oriented Design Pattern Detection in Object-Oriented Systems

AUTHOR: Lei Hu, B.Eng.

SUPERVISORS: Dr. Kamran Sartipi

NUMBER OF PAGES: xiv, 102.

Abstract

Identifying design pattern instances within an existing software system can support understanding and reuse of the system functionality. Moreover, incorporating behavioral features through task scenario into the design pattern recovery would enhance both the scalability of the process and the usefulness of the design pattern instances. In this context, we present a novel method for recovering design pattern instances from the implementation of system behavioral features through a semi-automatic and multi-phase reverse engineering process.

The proposed method consists of a feature-oriented dynamic analysis and a two-phase design pattern detection process. The feature-oriented dynamic analysis works on the software system behavioral features' run-time information and produces a mapping between features and their realization at class level. In the two-phase design pattern detection process, we employ an approximate matching and a structural matching to detect the instances of the target design pattern described in our proposed Pattern Description Language (PDL), which is an XML-based design pattern description language. The correspondence between system features and the identified design pattern instances can facilitate the construction of more reusable and configurable software components. Our target application domain is an evolutionary development of software product line which emphasizes on reusing software artifacts to construct a reference architecture for several similar products. We have implemented a prototype toolkit and conducted experimentations on three versions of JHotDraw systems to evaluate our approach.

Acknowledgments

I would like to express my sincere appreciation to my supervisor Dr. Kamran Sartipi for his remarkable guidance and encouragement in all the time of my research. His overly enthusiasm and integral view on research and his mission for providing high-quality work has made a deep impression on me.

I would also like to appreciate my M.A.Sc. defense committee Dr. Tom Maibaum and Dr. Mark Lawford for their valuable comments which are useful for improving my work.

Lastly but not least, I would like to thank my family for their support and love. Special thanks to Sui Huang, Feng Xie, Siqian Mao for their support during my study and research.

Contents

Abstract iii

Acknowledgments iv

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Problem Description 2

1.2 Proposed Solution 3

1.2.1 Proposed Framework for Feature-Oriented Design Pattern Detection Approach 3

1.3 Thesis Contribution 7

1.4 Thesis Overview 8

2 Related Work 9

2.1 Design Pattern Detection 10

2.2 Concept Analysis 16

2.3 Dynamic Analysis 17

2.4 Software Product Line 18
2.4.1 An Evolutionary Development Process of A Software Product Line 18

3 Formal Definition of the Proposed Approach **20**
3.1 Terminology for Feature-Oriented Dynamic Analysis 20
3.2 PDL Representation of Design Pattern 22
3.2.1 Pattern Description Language (PDL) 23
3.2.2 Design Pattern Description in PDL 26
3.3 Modeling the Two-phase Design Pattern Detection Process . . . 28

4 Feature-Oriented Dynamic Analysis **30**
4.1 Feature-Specific Scenario Set Generation 31
4.2 Execution Traces Generation 32
4.3 Execution Pattern Extraction 34
4.4 Execution Pattern Analysis 35
4.4.1 Concept Analysis 36
4.4.2 Execution Pattern Analysis Using Concept Analysis . . . 38

5 Design Pattern Detection **40**
5.1 Pre-processing: Structural Information Extraction 41
5.2 Overview of the Proposed Two-phase Design Pattern Detection Process 41
5.3 Approximate Matching 44
5.3.1 Attribute Vector 45
5.3.2 Proposed Similarity Function 46
5.3.3 Algorithm 47

5.4 Structural Matching 49
5.4.1 Algorithm 49
5.5 Algorithm Complexity Analysis 54
5.5.1 Complexity of Approximate Matching Algorithm 55
5.5.2 Complexity of Structural Matching Algorithm 55
5.6 An Example 57

6 Case Study **62**
6.1 Experimental Platform 63
6.2 Subject System 63
6.3 Feature-Oriented Dynamic Analysis 64
6.3.1 Feature-Specific Scenario Set Generation 64
6.3.2 Execution Pattern Extraction 65
6.3.3 Execution Pattern Analysis 68
6.4 Design Pattern Detection 69
6.4.1 Pattern Repository 70
6.4.2 Results of Design Pattern Detection 70
6.5 Discussion 73

7 Conclusion and Future Work **83**
7.1 Discussion 84
7.2 Future Work 86

Bibliography **86**

List of Figures

1.1 The proposed framework for feature-oriented design pattern detection. 4

2.1 An Iterative and Evolutionary Development Process for a Software Product Line [15]. 19

3.1 Class diagram of a target design pattern. 26
3.2 PDL representation of the target pattern in Figure 3.1. 27

4.1 A part of an execution trace collected by TPTP. 34

5.1 The proposed framework for feature-oriented design pattern detection. 43
5.2 Class diagram of a target pattern. 46
5.3 A model of source-class cluster. 55
5.4 Class diagram and PDL representation of Bridge design pattern. 60
5.5 Class diagram of search space $\mathcal{SP}_1$. 61

6.1 Average execution trace size of selected scenarios of the 10 features in three versions of JHotDraw systems. 66

6.2 Concept lattice representation of the features and classes in JHotDraw 5.1. 68
6.3 Concept lattice representation of the features and classes in JHotDraw 6.0b1. 69
6.4 Concept lattice representation of the features and classes in JHotDraw 7.0.7. 70
6.5 Class diagram and PDL representation of Bridge design pattern. 75
6.6 A Strategy design pattern instance of feature *Drawing a Polygon* in JHotDraw 5.1. 76
6.7 Adapter design pattern instances related to feature Drawing a Polygon in JHotDraw 5.1, 6.0b1 and 7.0.7. 82

List of Tables

2.1 Design Patterns Classification by Purpose [20]. 11

5.1 Combinations of matched source classes of depth1-classes 58

5.2 Identified instances of Bridge design pattern base on one combination 58

6.1 Statistics of the three versions JHotDraw systems. 63

6.2 Results of execution trace extraction and execution pattern mining for 10 features of three versions of JHotDraw systems. . . . 67

6.3 Results of feature-specific classes assignment for 10 features of JHotDraw 5.1. 71

6.4 Results of feature-specific classes assignment for 10 features of JHotDraw 6.0b1. 72

6.5 Results of feature-specific classes assignment for 10 features of JHotDraw 7.0.7. 74

6.6 Results of identified Adapter design pattern instances and related features in JHotDraw 5.1 system 76

6.7 Results of identified Bridge design pattern instances and related features in JHotDraw 5.1 system 77

6.8 Results of identified Observer design pattern instances and related features in JHotDraw 5.1 system 77
6.9 Results of identified Strategy design pattern instances and related features in JHotDraw 5.1 system 78
6.10 Results of identified Adapter design pattern instances and related features in JHotDraw 6.0b1 system 79
6.11 Results of identified Strategy design pattern instances and related features in JHotDraw 6.0b1 system 80
6.12 Results of identified Adapter design pattern instances and related features in JHotDraw 7.0.7 system 80
6.13 Results of identified Strategy design pattern instances and related features in JHotDraw 7.0.7 system 81
6.14 Results of identified Bridge design pattern instances and related features in JHotDraw 7.0.7 system 81

7.1 Comparison of several design pattern detection techniques . . . 85

Chapter 1

Introduction

The role and function of software systems in modern industrial organizations are becoming more and more vital. In order to sustain their competitive position in industry, these organizations need customizable software systems with high quality, high reusability and short time-to-market.

Software product line is a solution to help these organizations to fulfill their goals. A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [15]. This set of software-intensive systems are developed based on a reference architecture, which consists of common and variable parts. The variable parts can be modified to satisfy the various needs from the market which can improve the quality and efficiency of software development.

Typically, an evolutionary development of a software product line starts from reverse-engineering a number of similar existing systems. The reverse engineering activities are conducted to identify the important behavioral *features* which have a high reuse potential and to locate their corresponding software

artifacts which can be reused in the software product line [12]. In the context of this thesis, a *feature* is an observable unit of software system behavior that describes a single functionality in the software system. For example, in a graphic drawing tool, JHotDraw [3], the operations "draw a rectangle", "move a rectangle" and "edit a rectangle" are three different features.

Design pattern recovery is one of the reverse engineering activities to support the construction of the software product line. Design patterns represent a common solution to a recurring design problem in object-oriented software systems [20]. Consequently, knowing the applied design patterns can enable us to understand the software system implementation at the design level and provide the ground for further improvement [23]. Moreover, it can help us to decide whether the software artifacts can be reused under the new requirement of the software product line. Therefore, identifying the design patterns instances applied in the existing systems can help both comprehension and reuse of the existing systems during the software product line construction process. This thesis aims to present a novel reverse engineering approach which combines *feature analysis* with *design pattern recovery* to support an evolutionary development of a software product line.

1.1 Problem Description

Over the last decade, software product lines have proven to be one of the most promising software development paradigms. Software product line aims at designing and implementing a family of products sharing some common core features. By providing a reference architecture for the development of similar products, software product line can improve the efficiency of software

development to satisfy the specific needs of the market.

A prerequisite for an evolutionary development of a software product line is understanding the key functionalities of the existing systems using reverse engineering techniques. Design patterns represent a high level of abstraction in OO design. Therefore, identifying the applied design pattern instances within the implementation of the existing systems can help with the comprehension of the adopted solutions for the systems. Furthermore, incorporating the system behavioral features into the design pattern recovery process not only enhance the scalability and the objectiveness of the process, but also improve the reusability of software artifacts during the software product line construction.

Based on the above discussion, we define the problem of this thesis as:

> *devising a method and the supporting tools for recovering the instances of the design patterns which are represented by a high level pattern description method, from the implementation of software system behavioral features.*

1.2 Proposed Solution

In this thesis, we propose a novel reverse engineering framework which combines feature-oriented dynamic analysis with a two-phase design pattern detection process to identify the instances of design patterns for software behavioral features.

1.2.1 Proposed Framework for Feature-Oriented Design Pattern Detection Approach

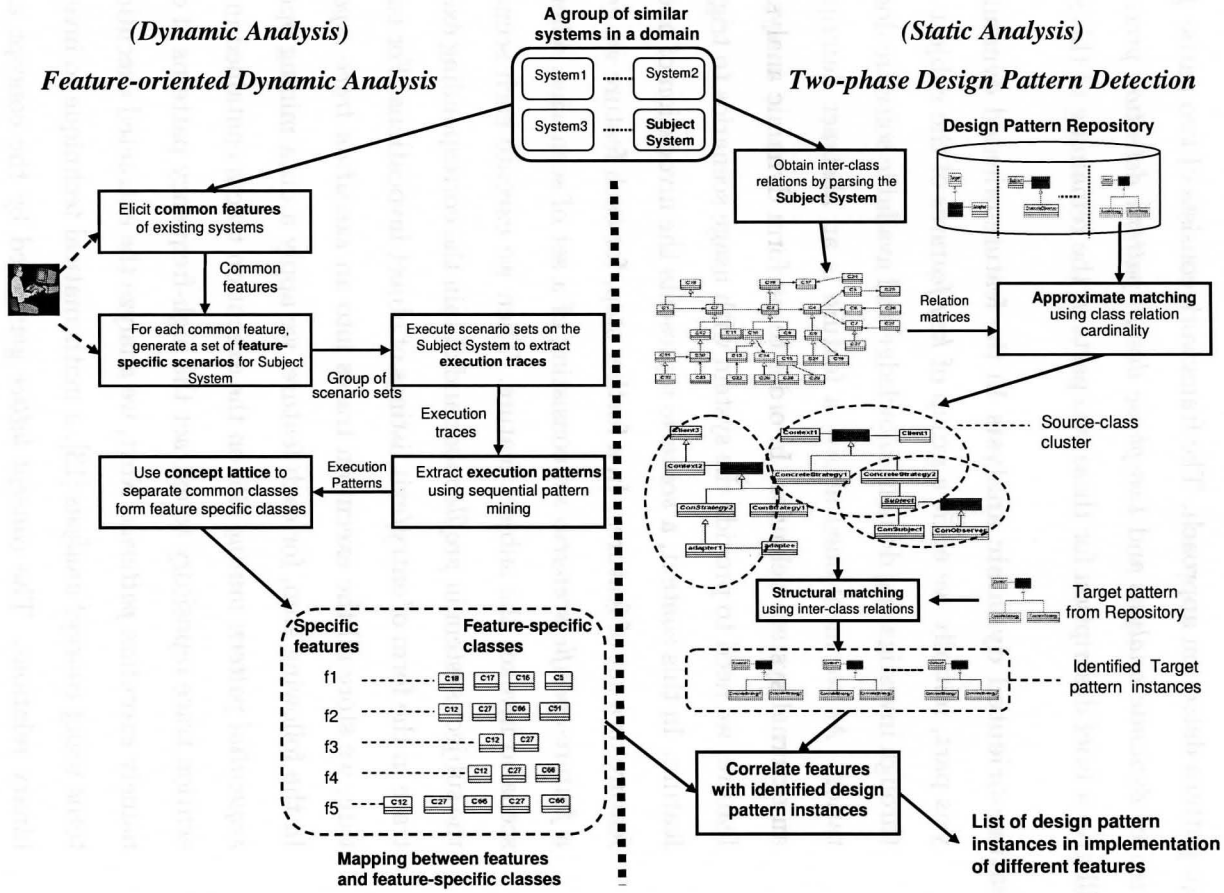


Figure 1.1: The proposed framework for feature-oriented design pattern detection.

Figure 1.1 illustrates the framework of the proposed feature-oriented design pattern detection approach. The framework consists of two parts: *feature-oriented dynamic analysis* and *two-phase design pattern detection process.* We will give a brief description for these two parts in the remaining of this section.

Feature-oriented dynamic analysis In the feature-oriented dynamic analysis part, initially we elicit a group of *key features* of the subject system through investigating domain knowledge and available software documentation. As we mentioned above, a feature is an abstract description of an external system behavior. In order to perform dynamic analysis for a feature, we need to provide the system with usage scenarios to trigger the feature. In this context, a *scenario* represents the invocation of one single feature or a set of features. In a further step, for each feature we generate a *feature-specific scenario set* consisting of a set of scenarios, where each scenario invokes the subject feature. Then we execute each scenario on the subject system in *profile mode* and obtain the corresponding execution trace in the form of entry/exit listings of object invocations. For each feature, we store all the execution traces into an *execution trace repository.* In the following step, for each feature, we apply a data mining operation *sequential pattern mining* [36] on the execution traces contained in its execution trace repository to extract the high-frequency patterns of classes, namely *execution patterns.* Next, we analyze the extracted execution patterns using *concept analysis* [13], a mathematical technique to investigate binary relations. The *concept lattice* generated by the concept analysis allows us to collect the corresponding classes that exclusively contribute in implementing each feature, namely *feature-specific classes.* Thus, we

obtain a mapping between features and their feature-specific classes as the result of the feature-oriented dynamic analysis.

A two-phase design pattern detection process In this process, given a repository of *target design patterns* represented by the proposed Pattern Description Language (PDL) and the extracted system structural information, we perform the two-phase design pattern detection process to identify all the instances of the target design patterns.

The two-phase design pattern detection process consists of *approximate matching* and *structural matching.*

I) *approximate matching* identifies the design pattern instances of the given target design pattern at a coarse-grained level. As we mentioned above, each target design pattern in the pattern repository is represented in PDL. In the PDL representation, we specify a *main-seed class* which is a center-role class in the target design pattern. Moreover, we name those classes directly related to the main-seed class as *depth1-classes.* Those classes which are not directly related to the main-seed class are denoted as *depth2-classes.* We will give the formal definitions of main-seed class, depth1-class and depth2-class in Chapter 3. By applying an *approximate similarity function* to all the *source-classes* in the search space, we collect the appropriate candidates for the main-seed class. For each candidate of the main-seed class, we grow it by adding all the related source-classes within two depths to generate a source-class cluster which contains one or more candidate design pattern instances.

II) *structural matching* receives the list of source-class cluster obtained

from the approximate matching and searches each source-class cluster to find all the structurally matched design instances of the given target design pattern. The structural matching is mainly composed of two steps: $Depth1Matching$ and $Depth2Matching$, which are used to find the candidate source-classes for the depth1-classes and depth2-classes, respectively.

The detailed description of these two parts of the proposed framework will be discussed in Chapters 4 and 5.

1.3 Thesis Contribution

This thesis presents a methodo which incorporates feature analysis into design pattern recovery to recover design pattern instances from the implementation of system behavioral features. The approach proposed in this thesis contributes the followings:

- proposing a novel framework that incorporates feature-oriented dynamic analysis of a software system with static and template-based design pattern detection process to identify the design pattern instances in the software feature's implementations;
- presenting a two-phase design pattern detection process, where an approximate matching operation generates a reduced search space for a structural matching operation, thus to reduce the complexity of the design pattern detection process;
- providing a new design pattern representation, PDL (Pattern Description Language), which enables users to describe the structural information

of design patterns efficiently and conveniently, even to define their own patterns (user-specified patterns) other than those design patterns in [20];

- enhancing the previous work [33] on mapping software behavior features to source code, including extending the target systems to object-oriented systems and improving the accuracy of the obtained running information by using Eclipse Test and Performance Tools Platform (TPTP) [4] as our profiling tool.

1.4 Thesis Overview

The remaining Chapters of this thesis are organized as follows:

Chapter 2: summarizes the related work in the area of dynamic analysis and design pattern detection.

Chapter 3: presents the definition of relevant terminology that we use throughout this thesis.

Chapter 4: describes the feature-oriented dynamic analysis for feature implementation location.

Chapter 5: gives a detailed description of the two-phase design pattern detection process.

Chapter 6: discusses the experimental results of the applying the proposed approach to three versions of JHotDraw systems.

Chapter 7: draws a conclusion for the whole thesis and gives the future work.

Chapter 2

Related Work

Design Patterns are descriptions of communicating classes and objects that provide a common solution to a general and recurring software design problem in a particular context [20]. Since design patterns were proposed by Gamma et al. in 1993, they have been widely applied by software engineers in object-oriented software development and proved to be a useful technique of software engineering in the following aspects:

- enhancing software artifacts reuse
- helping understand adopted solution of existing software systems
- providing a communication platform between software developers
- facilitating software documentation

According to the taxonomy proposed in [20], design patterns can be classified into the following three categories according to their purposes.

- Structural design pattern

This kind of design patterns mainly deal with the composition of the pattern classes. Through defining class hierarchies and different inter-class relations, structural design patterns compose the pattern classes to form larger structures with new functionality. In structural design patterns most features are described with the declarations of the operations and attributes.

- Behavioral design pattern

  Behavioral design patterns characterize the ways in which classes or objects interact and distribute responsibility [20]. They are concerned with not just patterns of objects or classes but also the patterns of communication between them, which can help us define the communication between objects and how the flow is controlled in a program.

- Creational design pattern

  The creational design patterns focus on object creation mechanisms. In some particular context, the basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by controlling the type of the object and its multiplicity to create objects in a manner suitable to the situation.

Table 2.1 shows classification of the 23 Design patterns presented in [20] based on their purposes.

2.1 Design Pattern Detection

In this section we discuss some related works that have been proposed to address the problem of design pattern detection. Based on the characteristic of these

Structural	Behavioral	Creational
Adapter	Interpreter	Abstract Factory
Bridge	Template Method	Builder
Composite	Chain of Responsibility	Factory Method
Decorator	Command	Prototype
Facade	Iterator	Singleton
Proxy	Mediator	
	Memento	
	Flyweight	
	Observer	
	State	
	Strategy	
	Visitor	

Table 2.1: Design Patterns Classification by Purpose [20].

techniques we can classify these different approaches into following three major categories.

Design pattern detection based on class structure This kind of design pattern detection technique is performed to identify the design pattern instances which have the same pattern class structures with the target design patterns. There are several proposed approaches in this category:[11, 29, 37, 31, 45, 27, 14, 44].

Kramer et al. [29] proposed the Pat system, which can detect the instances of structural design patterns from C++ source code. By using an object-oriented case tool, Paradigm Plus, Pat extracts the static information from the C++ head files and converts it into PROLOG facts. The extracted information includes class names, attribute names, method

names, inheritance, association and aggregation relations. On the other hand, design patterns are defined by using the PROLOG rules. The pattern detection process is accomplished by applying the PROLOG queries. However, due to the limitation of the extractor provided by Paradigm Plus, some structural information is missed, such as the category of class (abstract or concrete) and delegation. As a consequence, the precision of the Pat system is not satisfied. Comparing with the Pat system which is only focused on the static analysis, our proposed approach combines dynamic analysis with static analysis, thus achieves a more accurate result.

In [37], Nija Shi et al. proposed a new, fully automated approach to discover design patterns from Java source code. They focus on detecting the structural and behavior-driven patterns using only static program analysis. A structural design pattern can be identified by the inter-class relationships which are obtained by parsing the source code. To detect a behavior-driven design pattern, first they apply inter-class analysis to obtain the candidate pattern set which conforms to the structural aspects of the target pattern. The resulting candidate set is taken as the input for the following behavioral analysis. To address the problem of behavioral aspects of detecting design pattern process, the authors use data-flow analysis on Abstract Syntax Tree (AST) in terms of basic blocks. They do not analyze the whole AST of the method body. They generate a control-flow graph (CFG) for the method body by linking together the basic blocks based on execution flow. Then using the static analysis on the CFG, they can verify the behavioral aspects of the candidate pattern is consistent with the behavior of the target pattern. They also present a

tool, PINOT (Pattern INference and recOvery Tool), which implements this new approach. Comparing with our presented toolkit, PINOT hard-codes all design pattern detection algorithms, therefore it can not allow user to define their desired pattern.

A similar approach was presented in [11], the authors described a method based on a multi-stage recovering strategy using Design Pattern Markup Language (DPML) and Columbus [32] to detect design patterns from C++ source code. First, an internal representation, Abstract Semantic Graph (ASG) is extracted from the C++ source code using the Columbus system. Next, pattern descriptions of the target patterns are loaded in a form of Design Pattern Markup Language (DPML). Finally, the multi-stage recovering process match source classes with pattern classes and check whether they are related in a way which is described in DPML. The main advantage of this work is that it provides design pattern instance detection using a user-friendly language for design pattern description. However, because of the language limitation of Columbus, this method can only be applied for the C++ source code. Similar with our proposed Pattern Description Language (PDL), DPML also provides an easy way for the users to define their own patterns to suit their needs. However, the proposed PDL classifies the pattern-class into *main-seed class*, *depth1-class*, *seed-depth1-class* and *depth2-class* to efficiently describe the structure of a design pattern.

In [31] Lucia et al. proposed a two phase approach to the recovery of structural design pattern. In the first phase, a coarse-grained analysis, which was proposed in the previous work [16], was performed to reduce the num-

ber of the candidate patterns by analyzing the class diagram structure. In the second phase a fine-grained source code analyze was used to check whether the identified the candidate patterns are real patterns by verifying the inter-class relations between the classes of the candidate patterns, which can eliminate the most of the false positive pattern instances.

Design pattern detection based on metrics In this type of approaches, software metrics are used to characterize pattern classes. Typical techniques include [7], [23] and [28].

In [7] Antoniol et al. presented an automatic and conservative multi-stage reduction approach, which uses software metrics to reduce the search space and then utilizes the structural information between classes to perform the extract structural matching. A target design pattern is represented as tuple of classes and relations among these classes. By orderly applying class level metrics constraints, shortest path constraints and structural constraints on the candidate sets, this approach avoid the combinatorial explosion in checking all possible class combinations while determining pattern constituents' candidate sets, which reduces the search space greatly. For the intent of language independence, by using AOL extractor, source code and design are converted to an intermediate representation, called Abstract Object Language (AOL). Then the AOL software artifacts are parsed to get the Abstract Syntax Tree (AST) from which software metrics and structural information can be extracted. By orderly applying class level metrics constraints, shortest path constraints and structural constraints on AST, instances of design patterns are identified.

Yann-Gael et al. [23] proposed a technique, named fingerprinting, for detecting design patterns from source codes. To reduce the search space, they use software metrics (such as size, cohesion, and coupling) and a machine learning algorithm to fingerprint the role of each class participating in the detected design pattern. In other words, they use a set of metric values to characterize the classes playing in a given role in the design pattern. In the following phase, actual pattern instance are found with structural matching. The efficiency of such an algorithm depends strongly on the learning samples that compose the repository of design motif roles.

Comparing with above two approaches based on software metrics, we use a set of attribute values rather than to fingerprint one class but to perform an approximate matching to identify a candidate pattern (a set of classes).

Design pattern detection based on matrices This kind of design pattern detection approaches store the system's inter-class relations into matrices, similarly, the structural information of target patterns are also store into matrices. Thus, the pattern matching process is accomplished by matrices matching.

Nikolaos et al. [43] presented an automatic approach which use a similarity score algorithm to detect design patterns. Different structural inter-class relations are extracted and stored into corresponding matrices. Similarly, the structural information of target design patterns are also represented by matrices. The detection of design patterns is accom-

plished by calculating the similarity score between the matrices of system and those of target design patterns. They present a tool and conduct experiments on three Java open source systems: JHotDraw 5.1, JRefactory 2.6.24 and JUnit 3.7.

The authors of [17] also proposed a novel approach based on matrix to detect design patterns from source code. The inter-class relations is represented in a matrix and the value of each cell represents the different relations among the classes. The structure of each design pattern is also represented in matrix. The discovery of design patterns from source code becomes matching between the two matrices. In contrast to the approach in [43], they store all the inter-class relation information into a single matrix using prime numbers. The authors developed a design pattern discovery tool, DP-Miner, which implemented the novel approach and conducted several experiments on the open-source systems, including JUnit, JEdit, JHotDraw, and Java.AWT.

2.2 Concept Analysis

Concept analysis [21] is a branch of lattice theory that allows us to identify meaningful groupings of *objects* that have common *attributes.*

The sets of objects, attributes and binary relations between them are known as *context*, and the groupings based on the common attributes of the objects are named as *concepts.* Mathematically, concepts are maximal collections of objects sharing common attributes.

The mathematical concept analysis was first introduced by Birkhoff in 1940 [13]. In recent years, concept lattice analysis has been introduced to the

field of reverse engineering. It is used to recover class hierarchies in [10, 40]. In [30, 38, 39] concept lattice analysis is employed to identify modules from legacy source code. In dynamic analysis it is used to derive a feature-component correspondence [19, 36]. Moreover, it also helps to detect instances of design patterns in [9, 42].

2.3 Dynamic Analysis

In general, reverse engineering techniques can be categorized into (1) static analysis, i.e., by analyzing the source code, (2) dynamic analysis, i.e., by examining the programs behavior, or (3) a hybrid of both.

Static information obtained by static analysis describes the structure of the software, while dynamic information describes software system run-time behavior. To fully understand existing software both static and dynamic information need to be extracted [41].

In the context of reverse engineering of object-oriented systems, due to polymorphism and late-binding, static analysis is often inaccurate with regard to the actual behavior of the application [46]. Dynamic analysis, however, is able to obtain a real information of the program's run-time behavior. Therefore, dynamic analysis takes an important role in the field of object-oriented reverse engineering.

In [47, 26, 25, 24], through analyzing the execution traces of the software systems, frequently occurring interaction patterns between classes i.e., behavioral design models, are extracted, which can help the program comprehension process.

Many other researchers use run-time execution traces to address the prob-

lem of feature location, i.e., locating low-level system components that implement a specific software feature [19, 22, 18, 34, 8].

2.4 Software Product Line

A Software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission [15]. The similar products are developed based on a reference architecture, which consists of common and variable parts. The variable parts can be modified to satisfy the various needs from the market which can improve the quality and efficiency of software development.

Over the last decade, software product line has been proven to be one of the most promising software development paradigms. Nowadays many organizations are investing in software product line for the following reason: Through designing and implementing a family of products sharing some common core features, software product line allows those organizations to improve the quality and efficiency of software development, accelerate the process of introduction of new products, reduce the complexity and cost of development and maintenance, and manage the product variations needed for markets.

2.4.1 An Evolutionary Development Process of A Software Product Line

An evolutionary process of development of a software product line architecture is described in Figure 2.1. The evolutionary process of building a software product line starts from reverse-engineering the legacy software systems. In a further step, by analyzing and comparing the new requirements to the former

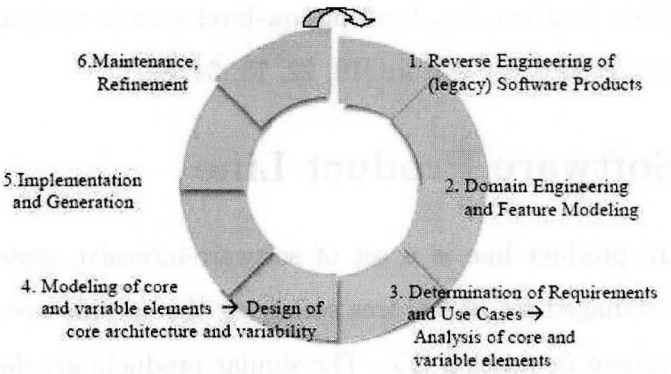


Figure 2.1: An Iterative and Evolutionary Development Process for a Software Product Line [15].

ones, the software product line requirements is determined. Finally, during the phases 4-6 the reference architecture of software product line architecture is designed and implemented. If there is a need for a new costumer requirement, the process goes to a new iteration.

Chapter 3

Formal Definition of the Proposed Approach

In this chapter, we present the definitions of relevant terminology that we use throughout this thesis to describe the *feature-oriented dynamic analysis* and the *two-phase design pattern detection* of our proposed framework. In Section 3.1, the definitions of *feature*, *scenario*, *feature-specific scenario set* and *feature-specific class* are presented. In Section 3.2, we introduce Pattern Description Language (PDL), a novel representation of design pattern. Finally, in Section 3.3, we present a brief description of the two-phase design pattern detection process using the defined notations. A thorough formal specification of the process is out of scope of this thesis.

3.1 Terminology for Feature-Oriented Dynamic Analysis

In this section, we present the definitions of the relevant terms that we use within the context of the feature-oriented dynamic analysis presented in our proposed approach.

Feature: A *feature* ϕ, is a single functionality of the subject software system. For example, in a graphic drawing software system, the operation to draw a rectangle can be considered as a feature. The set of all the features in the software system is denoted as Φ.

Scenario: A *scenario* represents a sequence of features of the subject software system. Formally, a *scenario* s is a sequence of features $\phi \in \Phi$, hence $s = [\phi_1, \phi_2, ..., \phi_n]$. For instance, a scenario for drawing a rectangle and moving the rectangle includes two features, "drawing a rectangle" and "moving a figure".

Feature-specific scenario set: A *feature-specific scenario set* S_{S_ϕ} of a feature ϕ is a set of scenarios S_ϕ, each of which contains the specific feature ϕ. As an example, for the feature "move" in a graphic drawing software system, the following two scenarios form a feature-specific scenario set.

1 *start, draw a rectangle,* ***move,*** *exit*

2 *start, draw a ellipse,* ***move,*** *exit*

Execution trace: Let C^s be the set of all classes in the software system, for a scenario s, the *execution trace* T^s of the scenario s is a sequence of classes in C^s invoked for executing this scenario.

Execution trace repository: An *execution trace repository* R_{S_ϕ} of a feature ϕ is a set of execution traces obtained by executing each scenario s in the feature-specific scenario set S_{S_ϕ}.

Execution pattern: For a feature ϕ, an *execution pattern* ep_ϕ is a contiguous

sequence of classes which is supported by at least *MinSupport* number of execution traces in the execution trace repository R_{S_ϕ}. An execution pattern ep_ϕ is supported by an execution trace t iff ep_ϕ is a subsequence of t.

Execution pattern repository: An *execution pattern repository* R_{ep_ϕ} of a feature ϕ is all the execution patterns obtained based on the definition of execution pattern above.

Feature-specific class: For a feature ϕ, a *feature-specific class* c_ϕ is a class existing in the execution trace repository R_{S_ϕ} which exclusively corresponds to the subject feature ϕ within the feature-specific scenario set S_{S_ϕ}.

3.2 PDL Representation of Design Pattern

Design pattern describes a common solution to a general and recurring software design problem in terms of classes and their communications. Therefore a generic representation of a design pattern p consists of a set of *pattern-classes* and *inter-class relations*. A pattern-class is specified as a participating class in the design pattern and an inter-class relation is a relation between pattern-classes (e.g. inheritance, association, etc.). Formally, a design pattern p can be represented as a tuple $< \mathcal{C}, \mathcal{R} >$, where $\mathcal{C}$ is a set of *pattern-classes* $\{c_1, ..., c_k\}$ and $\mathcal{R}$ is a set of *inter-class relations* among these pattern-classes.

Considering a generic representation $< \mathcal{C}, \mathcal{R} >$ of a design pattern p as a graph, where a vertex represents a pattern-class in $\mathcal{C}$ and an edge corresponds to an inter-class relation in $\mathcal{R}$, we give the following two definitions.

ShortestPath: For two pattern-classes c_i and c_j in $\mathcal{C}$, the shortest path from c_i to c_j, denoted as $ShortestPath(c_i, c_j)$, is the minimum number of inter-class relations traversed to reach c_j from c_i, regardless of the type of the inter-class relations [7].

Degree: The *Degree* of a pattern-class c_i in $\mathcal{C}$, denoted as $deg(c_i)$, is the number of the direct inter-class relations between c_i and all the other pattern-classes in the design pattern p.

To construct an efficient design pattern detection approach, we propose Pattern Description Language (PDL), an XML-based language, to describe the structure of a design pattern. The proposed PDL representation is based on the generic representation and provides an efficient and convenient way to describe structural aspect of a design pattern. It also allows users to modify pattern description to suit their needs or even to define their own patterns that they desire to discover.

In the remainder of this section, we first introduce PDL representation, then discuss how to describe a design pattern using PDL.

3.2.1 Pattern Description Language (PDL)

Similar with the design pattern generic representation, a design pattern PDL representation consists of a set of *pattern-classes* and *inter-class relations* as well. Moreover, to efficiently describe the structure of a design pattern, we classify the pattern-class into *main-seed class*, *depth1-class*, *seed-depth1-class* and *depth2-class*. An inter-class relation is a relation between pattern-classes including:

- *Inherit_From*
- *Inherited_By*
- *in_Association*
- *out_Association*
- *is_Abstract*

The starting point of the design pattern PDL representation is the *main-seed class*, which is the central-role pattern-class in the design pattern. The idea of main-seed class comes from the previous work on a pattern-based software architecture recovery [35]. Before giving its definition, we first introduce some fundamental concepts used in the definition of the main-seed class.

We observe that for each design pattern presented in [20], there exists at least one pattern-class which can reach any other pattern-classes in the design pattern within a shortest path value 2. We name this kind of pattern-class as *potential main-seed class*, whose formal definition is given below.

Potential main-seed class: In a design pattern $p = \langle \mathcal{C}, \mathcal{R} \rangle$, a potential main-seed class, denoted as c^{pms}, is a pattern-class $c^{pms} \in \mathcal{C}$ such that $\forall c_i \in \mathcal{C} \bullet ShortestPath(c^{pms}, c_i) \leq 2$. C^{pms} is referred to the set of all the potential main-seed classes in the design pattern p.

Taking into account the breath-first strategy adopted by our proposed design pattern detection approach, we select the potential main-seed class with the maximal degree as the main-seed class for the design pattern p. This can help reduce the search space during the further two-phase design pattern

detection process. The formal definition of the main-seed class is presented below.

Main-seed class: In a design pattern $p = \langle \mathcal{C}, \mathcal{R} \rangle$, a main-seed class, denoted as c^{ms}, is a potential main-seed class $c^{ms} \in C^{pms}$ such that $\forall c_i \in C^{pms} \bullet deg(c_i) \leq deg(c^{ms})$. The set of main-seed class is denoted as C^{ms}. If there exist more than one main-seed classes in C^{ms}, we choose one of them randomly.

Based on the definition of the main-seed class of the PDL representation, we give the definitions of *depth1-class*, *seed-depth1-class* and *depth2-class* as follows.

Depth1-class: A *depth1-class*, denoted as c^{d1}, is a pattern-class in $\mathcal{C}$ which has a direct relation with the main-seed class c^{ms}. Formally, in a design pattern $p = \langle \mathcal{C}, \mathcal{R} \rangle$, a depth1-class is a pattern-class $c^{d1} \in \mathcal{C}$ such that $(c^{d1}, c^{ms}) \in R \vee (c^{ms}, c^{d1}) \in R$. The set of all the depth1-classes in the design pattern p is denoted as C^{d1}.

Depth2-class: A *depth2-class*, denoted as c^{d2}, is a pattern-class in $\mathcal{C}$ which is not related to the main-seed class directly. Formally, in a design pattern $p = \langle \mathcal{C}, \mathcal{R} \rangle$,a depth2-classes is a pattern-class $c^{d2} \in \mathcal{C}$ such that $ShortestPath(c^{ms}, c^{d2}) = 2$. The set of all the depth2-classes in the design pattern p is denoted as C^{d2}.

Seed-depth1-class: A *seed-depth1-class* denoted as c^{sd1}, is a depth1-class which is related to at least one depth2-class.

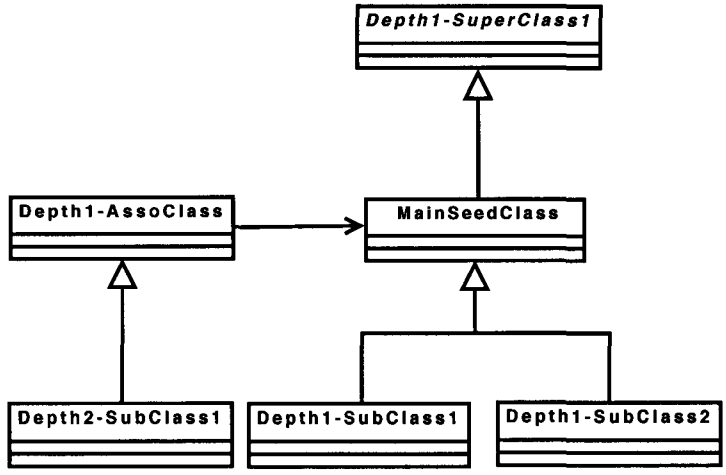


Figure 3.1: Class diagram of a target design pattern.

3.2.2 Design Pattern Description in PDL

In this section, we discuss how to describe a given design pattern with respect to the proposed PDL.

We use a breadth-first strategy to describe all the inter-class relations existing in the design pattern. In the first depth (depth1), starting with the main-seed class, we list all the involved inter-class relations of the main-seed class and the related depth1-classes in the item "Depth1" using the following syntax:

<inter-class relation> <depth1-class>*

In the second depth (depth2), we start with the seed-depth1 classes, each of which is related to one or more depth2-classes. Similarly, we list the involved inter-class relations of the seed-depth1-class and the related depth2-classes in

```
1  Begin-PDL
2    Pattern : TargetPattern
3      Main-seed class : MainSeedClass
4        Depth1 :
5          Inherit_From :
6            Depth1-SuperClass1
7          Inherited_By :
8            Depth1-SubClass1;
9            Depth1-SubClass2
10         in_Association :
11           Depth1-AssoClass
12       Depth2 :
13         Seed-Depth1: Depth1-AssoClass
14           Inherited_By :
15             Depth2-SubClass1
16       AbstractClasses :
17         Depth1-SuperClass1
18     End-Pattern
19   End-PDL
```

Figure 3.2: PDL representation of the target pattern in Figure 3.1.

item "Depth2" using the following syntax:

<seed-depth1-class> <inter-class relation> <depth2-class>*

In the item "AbstractClasses", we list all the abstract classes in the design pattern.

Figure 3.2 illustrates the PDL description of a target pattern in Figure 3.1. Line 2 indicates the name of the pattern. Each pattern in the pattern repository has a unique name. Line 3 specifies the main-seed class in the target pattern. Lines 4 to 11 describe the structural relations between main-seed class and all the depth1-classes. Lines 12 to 15 indicate the structural information

between the seed-depth1-class and the depth2-class in the pattern. Lines 16 to 17 list the abstract classes.

3.3 Modeling the Two-phase Design Pattern Detection Process

In this section, we use the aforementioned definitions to model the proposed two-phase design pattern detection process. According to the proposed framework presented in Chapter 1, the two-phase design pattern detection process consists of *approximate matching* and *structural matching* phases. The inputs of the process are a search space $\mathcal{SP}$ which consists of a set of source-classes and a target design pattern tp in PDL representation.

- **Approximate matching** searches the eligible candidates within the search space $\mathcal{SP}$ for the main-seed class c^{ms} of the target design pattern tp, and returns a list of source-class clusters CL, each of which contains a candidate of the main-seed class c^{ms}_{candi}.

- **Structural matching** identifies the structurally matched design pattern instances within each source-class cluster $cl \in CL$. It is composed of two steps: *Depth1Matching* and *Depth2Matching*. Depth1Matching receives a source-class cluster cl and collects the candidates for each depth1-class $c^{d1} \in C^{d1}$ to generate a set of combinations of matched source-classes of all the depth1-classes, denoted as CM^{d1}. Based on each combination $cm^{d1} \in CM^{d1}$, Depth2Matching collects the candidates for each depth2-class $c^{d2} \in C^{d2}$ to generate a set of combinations of matched source-classes of all the depth2-classes, denoted as CM^{d2}. Finally, we merge

the candidate of main-seed class c^{ms}_{candi}, the input combination of depth1-classes cm^{d1} and each combination $cm^{d2} \in CM^{d2}$ to obtain a complete identified design pattern instance .

Chapter 4

Feature-Oriented Dynamic Analysis

In the context of reverse engineering of object-oriented systems, due to polymorphism and late-binding, static analysis is often inaccurate with regard to the actual behavior of the application [46]. Dynamic analysis, however, is able to obtain real information on the software system's run-time behavior through analyzing its execution traces. Therefore, dynamic analysis takes an important role in the field of object-oriented reverse engineering.

In this thesis, we propose a feature-oriented dynamic analysis approach to locate the implementation of key features in object-oriented systems, which is an enhancement of the previous work presented in [33, 34]. This previous work is limited to the software systems written in procedural language, while our proposed approach extends its target system to object-oriented systems. Moreover, we choose Eclipse Test and Performance Tools Platform (TPTP) [4] as our profiling tool which enables us to collect the software system's execution information more conveniently and accurately.

The proposed feature-oriented dynamic analysis works on the run-time

execution traces of a set of subject features to locate the corresponding low-level system components that implement each feature. It consists of the following steps:

- Feature-specific scenario set generation.
- Execution traces generation.
- Extracting execution patterns from the execution traces.
- Execution pattern analysis

The remainder of this chapter is organized as follows. In Section 4.1 we discuss the feature-specific scenario set generation. In Section 4.2 we present the process of execution traces generation. In Section 4.3 we discuss the execution pattern extraction. Finally we give a brief description for execution pattern analysis in Section 4.4.

4.1 Feature-Specific Scenario Set Generation

According to the definition in chapter 3, a feature is a unit of software behavior that describes a single functionality in the subject software system. Given a subject feature ϕ, in order to locate its implementation classes, we first generate a feature-specific scenario set S_{S_ϕ} consisting of a set of scenarios S_ϕ, each of which contains the subject feature ϕ.

As an example, for the feature "move" in JHotDraw [3], a graphic drawing software system, we select the following scenarios to create a feature-specific scenario set.

1 *start, draw a rectangle,* ***move****, exit*

2 *start, draw a ellipse,* ***move****, exit*

3 *start, draw a polygon,* ***move****, exit*

4 *start, draw a line,* ***move****, exit*

5 *start, insert a image,* ***move****, exit*

These 5 scenarios share the feature "move" which is an operation to move a figure in JHotDraw.

The obtained feature-specific scenario set is taken as the input for the next step execution traces generation.

4.2 Execution Traces Generation

In this step, we use Eclipse Test and Performance Tools Platform (TPTP) to collect the execution traces generated by running the scenarios in the feature-specific scenario set. TPTP is an open platform which provides the services such as application monitoring, profiling and tracing. It includes a profiler that uses the Java Virtual Machine Profiler Interface (JVMPI) to report application events to the workbench. The examples of events include method entries, method exits, class loads and object allocations.

In order to only collect our desired events, we can define an effective filter set to indicate the classes whose running information is to be collected. The content of the filter set is a collection of rules that are applied by the JVMPI agent from top to bottom (higher rules have a higher precedence over the rules below them) [4]. Each rule has the following syntax:

<CLASS-NAME> <METHOD-NAME> <EXCLUDE | INCLUDE>

For example, to collect the execution traces of JHotDraw 5.1 system, we define a filter set which consists of the following two rules:

1. CH.ifa.draw* * INCLUDE
2. * * EXCLUDE

Rule 1 includes all classes with the prefix CH.ifa.draw and rule 2 excludes all other classes. Consequently, we are able to focus on the classes within JHotDraw 5.1 system and eliminate the library classes, e.g. Java system classes, from the execution traces.

For a subject feature ϕ, by running each scenario s_i within the feature-specific scenario set S_{S_ϕ} on the subject software system in a profiling mode, we obtain a corresponding execution trace t_i of the scenario s_i in the form of entry/exit listings of object invocations. We store all the execution traces into an *execution trace repository*, denoted as R_{S_ϕ}. Figure 4.1 presents a part of the execution trace of a scenario.

The large size of the execution traces is always a concern in dynamic analysis. Therefore, a preprocessing operation is applied to the execution traces to eliminate all the redundant object invocations caused by the program loops. The detailed description of the repetition elimination is presented in [33]. The trimmed execution traces are then fed into the next step, execution pattern extraction.

```
1   Enter CH/ifa/draw/util/PaletteButton
2   Enter CH/ifa/draw/standard/ToolButton
3   Exit CH/ifa/draw/standard/ToolButton
4   Enter CH/ifa/draw/standard/ToolButton
5   Enter CH/ifa/draw/util/PaletteIcon
6   Exit CH/ifa/draw/util/PaletteIcon
7   Enter CH/ifa/draw/util/PaletteIcon
8   Exit CH/ifa/draw/util/PaletteIcon
9   Exit CH/ifa/draw/standard/ToolButton
10  Exit CH/ifa/draw/util/PaletteButton
11  Enter CH/ifa/draw/util/PaletteButton
12  Enter CH/ifa/draw/standard/ToolButton
13  Exit CH/ifa/draw/standard/ToolButton
14  Enter CH/ifa/draw/standard/ToolButton
15  Enter CH/ifa/draw/util/PaletteIcon
16  Exit CH/ifa/draw/util/PaletteIcon
17  Enter CH/ifa/draw/util/PaletteIcon
18  Exit CH/ifa/draw/util/PaletteIcon
19  Exit CH/ifa/draw/standard/ToolButton
20  Exit CH/ifa/draw/util/PaletteButton
21  Enter CH/ifa/draw/util/PaletteButton
22  Exit CH/ifa/draw/util/PaletteButton
```

Figure 4.1: A part of an execution trace collected by TPTP.

4.3 Execution Pattern Extraction

In this section, we briefly describe the process of extracting highly repeated execution patterns of a subject feature from its execution trace repository. It receives an execution trace repository of a feature and returns a set of corresponding execution patterns for the feature.

As we mentioned in Chapter 3, an execution pattern is defined as a contiguous part in an execution trace that exists in certain number of execution

traces within the execution trace repository. Given a subject feature and the corresponding feature-specific scenario set, by applying a modified version of the sequential pattern mining algorithm [6] to the execution trace repository of the feature-specific scenario set, we obtain the execution patterns of the subject feature.

The sequential pattern mining algorithm consists of two main procedures: candidate two-items pattern generation (Procedure *cpGenerator*) and pattern extension (Procedure *DoExtend*) [33]. The procedure *cpGenerator* takes the execution trace repository as input and computes all eligible two-items patterns. The procedure *DoExtend* iteratively increases the length of the patterns obtained from Procedure *cpGenerator* to generate the eligible execution patterns. The detailed description of the algorithm is presented in [33].

This strategy generates the meaningful execution patterns for a subject feature, each of which consists of a set of classes implementing common functionalities within the feature-specific scenario set of the feature.

4.4 Execution Pattern Analysis

As we discussed in the previous two sections, we first generate a feature-specific scenario set S_{S_ϕ} consisting of a set of scenarios, each of which contains the subject feature ϕ. In a further step, by applying the aforementioned sequential pattern mining algorithm to the the execution trace repository R_{S_ϕ} generated by running the scenarios in the feature-specific scenario set S_{S_ϕ}, we obtain the execution patterns of the subject feature ϕ. However, since each scenario in the feature-specific scenario set S_{S_ϕ} contains not only the subject feature ϕ but also some other features, such as software initialization / termination, mouse

tracking, etc., which are needed to compose each scenario in the feature-specific scenario set S_{S_ϕ}. Therefore, the execution patterns extracted by the sequential pattern mining algorithm reflect both the implementation of the subject feature ϕ and the implementation of those common features [33]. In order to locate the implementation of the subject feature ϕ, we need to identify those classes which exclusively correspond to the subject feature ϕ within the feature-specific scenario set S_{S_ϕ}, we call this kind of class *feature-specific class.*

To address the above problem, we first examine a set of different features Φ in the subject software system, then extract the corresponding execution patterns for each of them. In a further step, we employ *concept analysis* to assign the corresponding *feature-specific classes* to each feature $\phi \in \Phi$.

In the remainder of this section, we first introduce concept analysis, then give a brief description of the execution pattern analysis using concept analysis.

4.4.1 Concept Analysis

Concept analysis is a mathematical technique that provides significant insights into a binary relation between *objects* and *attributes* to identify meaningful groupings of objects that have common attributes.

Context A *context* $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O}$ is a set of objects, $\mathcal{A}$ is a set of attributes and $\mathcal{R}$ is a binary relation between $\mathcal{O}$ and $\mathcal{A}$.

Common attributes For a set of objects $O \subseteq \mathcal{O}$, the set of *common attributes* σ is defined as:

$\sigma(O) = \{a \in \mathcal{A} \mid \forall o \in O \bullet (o, a) \in \mathcal{R}\}$.

Common objects For a set of attributes $A \subseteq \mathcal{A}$, the set of *common objects* τ is defined as:

$\tau(A) = \{o \in \mathcal{O} \mid \forall a \in A \bullet (o, a) \in \mathcal{R}\}$.

Concept A *concept* is a maximal collection of objects that possess common attributes, i.e., it is a grouping of all the objects that share a common set of attributes. In formal, a concept c is defined as a pair $< O, A >$, such that:

$A = \sigma(O) \wedge O = \tau(A)$.

O is said to be the *extent* of the concept c and A is said to be the *intent*.

Concept lattice Given two concepts $c_0 = (O_0, A_0)$ and $c_1 = (O_1, A_1)$, c_0 is a *subconcept* of c_1 if $O_0 \subseteq O_1$ (or, equivalently $A_1 \subseteq A_0$). The relation subconcept forms a partial order over the set of all concepts in a given context $\mathcal{C}$. Moreover, the set $\mathcal{L}$ of all concepts in the context $\mathcal{C}$ and the partial order form a concept lattice [13]:

$\mathcal{L}(\mathcal{C}) = \{(O, A) \in 2^{\mathcal{O}} \times 2^{\mathcal{A}} \mid A = \sigma(O) \wedge O = \tau(A)\}$

A concept lattice can be represented as an acyclic directed graph where a node represents a concept and an edge represents a subconcept relation [13]. To get more information from the concept lattice, we label the graph node with an attribute $a \in A$ whose represented concept is the most general concept that has a in its intent.

Each node (concept) N_i in the labeled graph representation of the concept lattice owns the following characteristics [18]:

- the contained objects of N_i are all the objects at and below N_i in the lattice.
- the contained attributes of N_i are all the attributes at and above N_i in the lattice.

The characteristics of the labeled concept lattice provide a significant insight into the structure of a relation between objects and attributes. Those attributes shared among most of the objects will appear in the upper region of the lattice, the nodes in the lower region of the lattice possess the attributes that are specific to the individual objects [33]. Therefore we can exploit this property to cluster the classes in the extracted execution patterns.

4.4.2 Execution Pattern Analysis Using Concept Analysis

In order to employ concept analysis to cluster the classes in the extracted execution patterns for a given group of features Φ, we define the context $\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{R})$ as follows:

- an object $o \in \mathcal{O}$ is a subject feature $\phi \in \Phi$, which is shared in the feature-specific scenario set S_{S_ϕ}.
- an attribute $a \in \mathcal{A}$ is a class c.
- a pair (ϕ,c) is in the binary relation $\mathcal{R}$ if class c participates in the execution patterns within S_{S_ϕ}.

The generated concept lattice by applying concept analysis to the defined context enables us to collect the feature-specific classes for each subject feature $\phi \in \Phi$. Since omnipresent classes are executed through almost every task scenario of the software system, these classes exist in the intent of almost every concept of the lattice and consequently appear in the upper region of the lattice. On the contrary, the feature-specific classes specific to each feature exist in the lower region of the lattice [33].

Chapter 5

Design Pattern Detection

Design pattern represents a high level of abstraction in object-oriented design. Thus, by retrieving its instances from software system, we are able to reveal the design decisions and adopted solutions of the system. Nevertheless, design pattern detection is not a trivial task, since we need to find all the possible design pattern instances whose structures are consistent with the target design pattern. Therefore, design pattern detection is a time-consuming, combinatorial problem with high complexity, easily resulting in combinatorial explosion [43]. To handle this issue, we present a novel and automated two-phase design pattern detection process which exploits the fact that there exists a main-seed class in each design pattern. As we discussed in Chapter 3, the main-seed class of a design pattern can reach any other pattern-classes within a shortest path value 2. Consequently, through finding the eligible candidates for the main-seed class, the system is partitioned into a group of source-class clusters, to which we apply the design pattern detection rather than to the entire system. This chapter is organized as follows. In Section 5.1 we give a brief introduction to system structural information extraction. An overview of the proposed two-

phase design pattern detection process is presented in Section 5.2. In Section 5.3 we discuss the process of *approximate matching*. A detailed description of *structural matching* is provided in Section 5.4. In Section 5.6 we use an example to illustrate the process of the proposed two-phase pattern detection approach.

5.1 Pre-processing: Structural Information Extraction

Before we conduct the two-phase design pattern detection process, we need to obtain the structural information of the subject system, i.e., inter-class relations. To accomplish this task, we employ a Java bytecode manipulation framework [1], which enables us perform a detailed analysis on the system's structure and extract the structural information we need in the further design pattern detection. The information retrieved includes inheritance, association and abstraction. We encode each kind of inter-class relation into an $n \times n$ matrix, where n is the number of the classes of the subject system and the columns and rows are all the classes in the system. Using matrices to represent the inter-class relations can facilitate us in the later approximate matching and structural matching.

5.2 Overview of the Proposed Two-phase Design Pattern Detection Process

According to the proposed framework presented in Figure 5.1, the process of the two-phase design pattern detection process consists of approximate matching and structural matching phases. In the approximate matching phase, through

identifying the eligible candidates for the main-seed class of the target design pattern, we reduce the search space for a target design pattern to a list of source-class clusters, each of which contains a candidate of the main-seed class. In the structural matching phase, we identify the structurally matched design pattern instances within the list of source-class clusters, which are obtained from the approximate matching. The structural matching phase is composed of two steps: $Depth1Matching$ and $Depth2Matching$, which are used to find the source-class candidates for the depth1-classes and depth2-classes, respectively. In Sections 5.3 and 5.4, we will give the detailed description of approximate matching and structural matching, respectively.

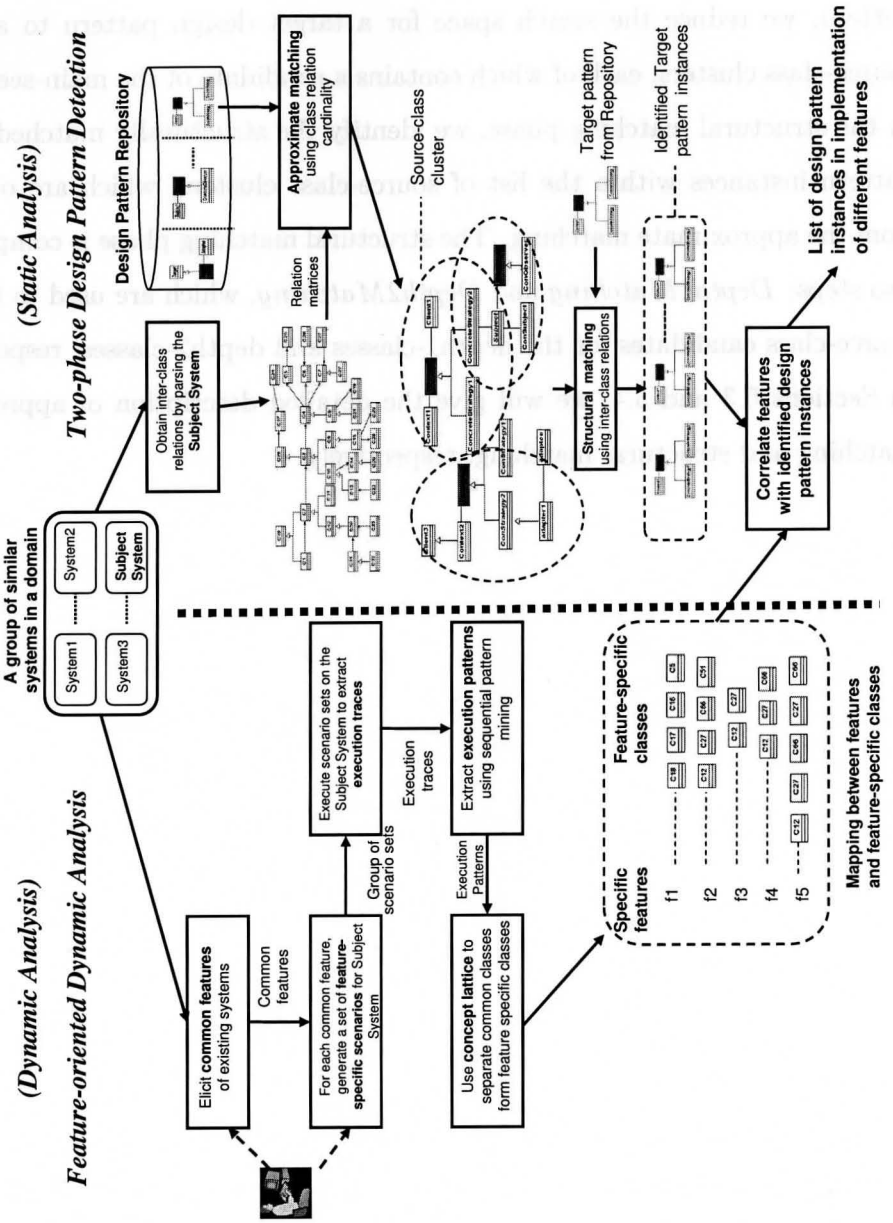


Figure 5.1: The proposed framework for feature-oriented design pattern detection.

5.3 Approximate Matching

One of the most important issues in design pattern detection is the sheer size of search space for a large software system [43]. As we mentioned in Chapter 3, A design pattern p can be represented as a tuple $< \mathcal{C}, \mathcal{R} >$, where $\mathcal{C}$ is a set of pattern-classes $\{c_1, ..., c_k\}$ and $\mathcal{R}$ is a set of inter-class relations among these pattern-classes. Given a subject system containing n source-classes and a target design pattern p consisting of k pattern-classes, a brute force approach to identify all the design pattern instances against the target design pattern p works in following steps. First it lists all the $n(n-1)(n-2)...(n-k+1)$ ordered combinations of the k pattern-classes within the n source-classes. Then it checks the validity of the set of inter-class relations $\mathcal{R}$ for each combination. Definitely, a combinational explosion would happen due to a great number of the source-classes within the subject system.

To address the combinational explosion problem, an approximate matching is proposed to find those source-class clusters containing potential candidate design pattern instances. First, through parsing the PDL (Pattern Description Language) representation of the target design pattern, we can obtain the specified main-seed class and a set of attributes (attribute vector) which characterizes the structural aspects of the design pattern with respect to its main-seed class. Secondly, an *approximate similarity function* is applied on all the source-classes in the search space to collect the appropriate candidates for the main-seed class. Finally, for each source-class candidate of the main-seed class, we grow it to form a source-class cluster by adding all the related classes within two levels. In summary, instead of matching all the pattern-classes at a time, the approximate matching only identifies those eligible candidates for the

main-seed class. This can reduce the search space to a number of source-class clusters, each of which contains one or more potential instances of the target design pattern.

5.3.1 Attribute Vector

To characterize a participating class in a design pattern in a structural perspective, we can specify an *attribute vector*, where each element is the number of inter-class relations of a certain type with which this class is involved.

The *attribute vector* includes the following items:

- the number of *Inherit_From* relation
- the number of *Inherited_By* relation
- the number of *in_Association* relation
- the number of *out_Association* relation
- the number of *is_Abstract* relation (0 or 1)[1]

As an example shown in Figure 5.2, the main-seed class "$MainSeedClass$" has one $in\_Association$ relation, one $Inherit\_From$ relation, and two $Inherited\_By$ relations, therefore the corresponding attribute vector $Attr\_Vec(MainSeedClass)$ is $[1, 2, 1, 0, 0]$.

[1]If a class is abstract, the item value is 1, otherwise 0.

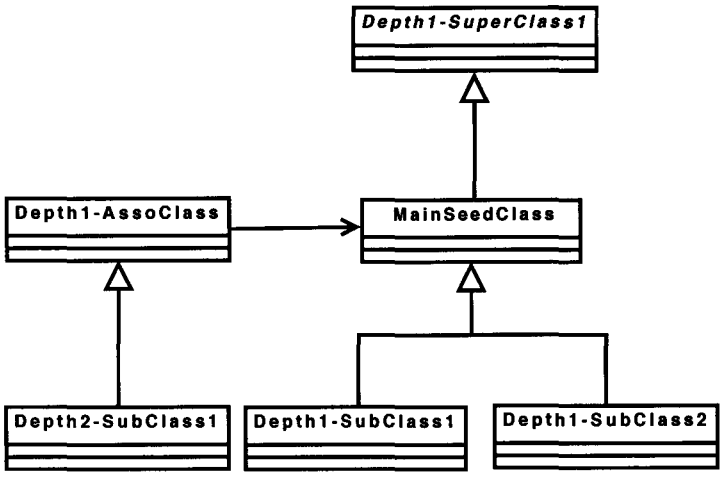


Figure 5.2: Class diagram of a target pattern.

5.3.2 Proposed Similarity Function

For a target design pattern *tp*, we apply a *similarity function* on all the source-classes within the search space to find all the eligible candidates for the main-seed class.

For the main-seed class c^{ms} of the target design pattern *tp*, we generate an attribute vector $Attr\_Vec(c^{ms}) = [a_1, ..., a_k]$, where $a_1, ..., a_k$ are the numbers of the inter-class relations $r_1, ..., r_k$, with which c^{ms} is involved.

Considering a search space $\mathcal{SP} = \{c_1, c_2, ..., c_n\}$, for each source-class $c_i \in \mathcal{SP}$, we also generate a corresponding *attribute vector* $Attr\_Vec(c_i) = [b_1, ..., b_k]$, where $b_1, ..., b_k$ are the numbers of the inter-class relations $r_1, ..., r_k$, with which c_i is involved.

Given the main-seed class c^{ms} of the target design pattern *tp* and a source-class $c_i \in \mathcal{SP}$, the approximate similarity function sim_{apx} is defined as:

$$sim_{apx}(Attr\_Vec(c_i), Attr\_Vec(c^{ms})) =$$

$$\begin{cases} \Delta(Attr\_Vec(c_i), Attr\_Vec(c^{ms})) & Attr\_Vec(c_i) \geq Attr\_Vec(c^{ms}) \\ 0 & otherwise \end{cases}$$

where $\Delta(Attr\_Vec(c_i), Attr\_Vec(c^{ms})) = 1 - \frac{\sum_{j=1}^{k}(Attr\_Vec_j(c_i) - Attr\_Vec_j(c^{ms}))}{\sum_{j=1}^{k}(max(Attr\_Vec_j(c_i), Attr\_Vec_j(c^{ms})))}$.

$Attr\_Vec(c_i) \geq Attr\_Vec(c^{ms})$ means that for any element a_k in the attribute vector $Attr\_Vec(c_i)$ is greater than or equal to b_k in the attribute vector $Attr\_Vec(c^{ms})$. In this context, the function sim_{apx} computes the approximate similarity value between the target design pattern (represented by the main-seed class c^{ms}) and the candidate instance pattern (represented by the main-seed candidate class c_i).

5.3.3 Algorithm

Algorithm 1 describes the process of the proposed approximate matching in detail. It receives a search space, the matrices of inter-class relations of the search space, a PDL representation of a target design pattern, and a cut-off threshold similarity value, and returns a list of source-class clusters, each of which contains an eligible candidate of the main-seed class.

The algorithm first generates an attribute vector for the main-seed class c^{ms} specified in the PDL representation of the target design pattern tp^{pdl}. In the second step, it iteratively generates an attribute vector for each source-class c_i within the search space $\mathcal{SP}$ and invokes function sim_{apx} to compute the similarity between this source-class c_i and the main-seed class c^{ms}. If the resulting similarity value is greater than or equal to the threshold similarity value tsh, then the source-class c_i is considered as an eligible candidate of

Algorithm 1: ApproximateMatching

Input:
$\mathcal{SP}$: search space (set of classes obtained from the subject system)
Ms: matrices of inter-class relations
tp^{pdl}: PDL representation of a target design pattern tp
tsh: cut-off threshold similarity value
Local Variable:
c^{ms}: main-seed class of design pattern tp
c_i: a source class in $\mathcal{SP}$
$Attr\_Vec(c_i)$: attribute vector of c_i
$Attr\_Vec(c^{ms})$: attribute vector of c^{ms}
cl: a source-class cluster which contains an eligible candidates of main-seed class
Result:
CL: list of source-class clusters containing eligible candidates of main-seed class

```
begin
    CL := [ ] ;
    c^ms := GetMainSeedClass(tp^pdl);
    Attr_Vec(c^ms) := ComputeAttrVector(tp^pdl) ;
    for class c_i ∈ SP do
        Attr_Vec(c_i) := ComputeAttrVector(c_i, Ms) ;
        if sim_apx(Attr_Vec(c^ms), Attr_Vec(c_i)) ≥ tsh then
            cl := GenerateCluster(c_i);
            CL := append(CL, cl);
end
```

the main-seed class c^{ms}. Finally, for each candidate of the main-seed class, the function *GenerateCluster*() is invoked to generate a source-class cluster surrounding the candidate of the main-seed class by adding all the related source-classes within two depths. The output of the algorithm is a list of source-class clusters, each of which contains an eligible candidate of the main-seed class c^{ms}, which is taken as the input of the following structural matching.

5.4 Structural Matching

The structural matching deals with the identification of all the instances of the target design pattern within a source-class cluster[2] of obtained from the aforementioned approximate matching.

5.4.1 Algorithm

Algorithm 2 describes the process of the structural matching. It receives a source-class cluster, a candidate of main-seed class, a PDL representation of a target design pattern, and the matrices of inter-class relations of the search space, and returns all matched design pattern instances within the source-class cluster.

The algorithm starts by collecting depth1-classes and depth2-classes of the target design pattern. This is accomplished by invoking functions *GetDepth1Classes*() and *GetDepth2Classes*(), respectively, which extract the corresponding depth1-classes and depth2-classes from the PDL representation of the target design pattern.

[2]Note that the number of the pattern-classes in a target design pattern is usually fewer than the number of the source classes in a source-class cluster, hence there may exist more than one matched pattern instances within the source-class cluster.

Algorithm 2: BFS-StructuralMatching

Input:
cl: a cluster of source-classes which contains an eligible candidate of main-seed class
c^{ms}_{candi}: a candidate source-class of main-seed class
tp^{pdl}: PDL representation of a target design pattern tp
Ms: matrices of inter-class relations

Local Variable:
cm^{d1}: a combination of matched source-classes of all the depth1-classes
cm^{d2}: a combination of matched source-classes of all the depth2-classes
CM^{d1}: set of all combinations of matched source-classes of all the depth1-classes
CM^{d2}: set of all combinations of matched source-classes of all the depth2-classes
C^{d1}: set of all depth1-classes in target design pattern
C^{d2}: set of all depth2-classes in target design pattern

Result:
R: set of identified design pattern instances

begin
  $R := \emptyset$;
  $C^{d1} := GetDepth1Classes(tp^{pdl})$;
  $C^{d2} := GetDepth2Classes(tp^{pdl})$;
  $CM^{d1} := Depth1Matching(tp^{pdl}, c^{ms}_{candi}, C^{d1}, cl, Ms)$;
  for $cm^{d1} \in CM^{d1}$ **do**
    $CM^{d2} := Depth2Matching(tp^{pdl}, cm^{d1}, C^{d2}, cl, Ms)$;
    for $cm^{d2} \in CM^{d2}$ **do**
      $R := R \cup merge(c^{ms}_{candi}, cm^{d1}, cm^{d2})$;
end

Algorithm 3: Depth1Matching

Input:
cl: a source-class cluster which contains an eligible candidate source class of main-seed class
Ms: matrices of inter-class relations
c^{ms}_{candi}: a candidate source-class of main-seed class
tp^{pdl}: PDL representation of a target design pattern tp
C^{d1}: set of all depth1-classes in target design pattern
Local Variable:
L_{tuple}: list of tuple < relation-type, depth1-class >
$t_{<rel,d1>}$: tuple < relation-type, depth1-class >
$M_{d1-srcclass}$: mapping between depth1-class and candidate source class set
C^{d1}_{candi} : set of candidate source-classes for a depth1-class
Result:
CM^{d1}: set of all combinations of matched source-classes of all the depth1-classes
begin
  $L_{tuple} := GetListofRelDepth1Tuple(tp^{pdl})$;
  $M_{d1-srcclass} = CreateandInitializeMapping(C^{d1})$;
  for $t_{<rel,d1>} \in L_{tuple}$ **do**
    $C^{d1}_{candi} = GetCandiSrcClass(c^{ms}_{candi}, t_{<rel,d1>}, cl, Ms)$;
    $AddtoMapping(M_{d1-srcclass}, d1, C^{d1}_{candi})$;
  $CM^{d1} = GenerateDepth1Combinations(M_{d1-srcclass})$;
end

In the second stage, the candidates are collected for each depth1-class. This is done by the function *Depth1Matching*() which is presented in Algorithm 3. In the function *Depth1Matching*(), first a mapping between each depth1-class and its candidate set is created, and each candidate set is initialized with an empty set. Secondly, by parsing the PDL representation of the target design pattern, a list of tuples < *relation-type, depth1-class* > are obtained, each of which represents an inter-class relation between the main-seed class and a depth1-class. Next, for each tuple < *relation-type, depth1-class* > we search the input source-class cluster to find those source-classes which have the same relation-type with the candidate of the main-seed class and add them to the candidate set of the depth1-class. Finally, by invoking the function *GenerateDepth1Combinations*(), a set of combinations of matched source-class of the depth1-classes are obtained. Each combination is taken as an input to the function *Depth2Matching*() to locate the eligible candidates for the depth2-classes.

In the next stage, for each combination of depth1-classes we search to find all the possible combinations of depth2-classes. This is accomplished by the function *Depth2Matching*() which is presented in Algorithm 4. The function *Depth2Matching*() receives a combination of depth1-classes and generates a set of combinations of depth2-classes. Likewise, it first generates a mapping between each depth2-class and its candidate set, and initialize each candidate set with an empty set. Secondly, by parsing the PDL representation of the target design pattern, we obtain a list of triples <*seed-depth1-class, relation-type, depth2-class*>. As we mentioned in Chapter 3, a seed-depth1-class is a depth1-class which is related to one or more depth2-classes. Each triple

Algorithm 4: Depth2Matching

Input:
cl: a source-class cluster which contains an eligible candidate source class of main-seed class
tp^{pdl}: PDL representation of a target design pattern tp
cm^{d1}: a combination of matched source-classes of all the depth1-classes
Ms: matrices of inter-class relations
C^{d2}: set of all depth2-classes in target design pattern
Local Variable:
L_{triple}: list of triple <seed-class, relation-type, depth2-class >
$t_{<s,rel,d2>}$: a triple < seed-class, relation-type, depth2-class >
$M_{d2-srcclass}$: mapping between depth2-class and candidate source class set
C^{d2}_{candi} : set of candidate source-classes for a depth2-class
Result:
CM^{d2}: set of all combinations of matched source-classes of all the depth2-classes
begin
  $L_{triple} := GetListofSeed\text{-}Rel\text{-}D2Triple(tp^{pdl})$;
  $M_{d2-srcclass} = GreateandInitializeMapping(C^{d2})$;
  for $t_{<s,rel,d2>} \in L_{triple}$ **do**
    $C^{d2}_{candi} = GetCandiSrcClass(t_{<s,rel,d2>}, c_{d1}, cl, Ms)$;
    $AddtoMapping(M_{d2-srcclass}, d2, C^{d2}_{candi})$;
  $CM^{d2} = GenerateDepth2Combinations(M_{d2-srcclass})$;
end

describes an inter-class relation between a seed-depth1-class and a depth2-class. Next, for each triple <*seed-depth1-class, relation-type, depth2-class*> we search the source-classes which have the same relation-type with the candidate of seed-depth1-class and add them to the candidate set of the depth2-class. Finally, through invoking the function *GenerateDepth2Combinations*(), we obtain a set of combinations of matched source-class of depth2-classes.

In the last stage, we merge each combination in the resulting set of combinations depth2-classes, the input combination of depth1-classes and the candidate of main-seed class to obtain a complete identified design pattern instance.

5.5 Algorithm Complexity Analysis

The design of a tractable algorithm is the most challenging part of the design pattern matching process. As we mentioned before, when the size of the search space becomes large, the brute force approach would result in a combinational explosion. To address this problem, a two-phase design pattern detection process is proposed to reduce the complexity of the matching process by finding those source-class clusters containing potential candidate design pattern instances first. The proposed approximate matching algorithm and structural matching algorithm of the two-phase design pattern detection process is a major contribution of this work. In this section, we discuss the complexity of the approximate matching algorithm and the structural matching algorithm, respectively.

5.5.1 Complexity of Approximate Matching Algorithm

In the proposed approximate matching algorithm, we identify the eligible candidates for the main-seed class of the target design pattern p by comparing the attribute vector of each source-class in the search space with that of the main-seed class. The running time of comparing the attributor vectors costs time $O(1)$, therefore, the running time of the approximate matching algorithm is $O(n)$, where n is the number of source-classes in the search space $\mathcal{SP}$.

5.5.2 Complexity of Structural Matching Algorithm

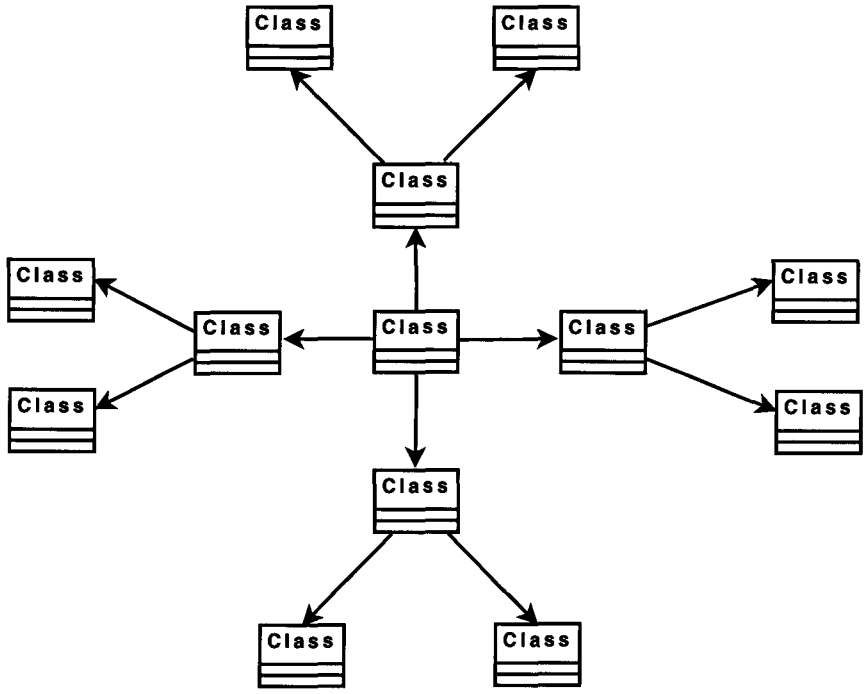


Figure 5.3: A model of source-class cluster.

The structural matching algorithm deals with the identification of all the

instances of the target design pattern within a reduced search space known as a source-class cluster obtained from the aforementioned approximate matching algorithm. It is composed of two steps: $Depth1Matching$ and $Depth2Matching$, which are used to find the source-class candidates for the depth1-classes and depth2-classes, respectively.

The complexity analysis of the structural matching is performed based on the following quantities:

- n: the number of source-classes in the search space $\mathcal{SP}$.
- k: the number of pattern-classes in the target pattern p.
- m: the number of source-class clusters obtained from the approximate matching.
- d_1: the number of depth1 source-classes in the source-class cluster model.
- d_2: the number of depth2 source-classes which are related to a depth1 source-class in the source-class cluster model.
- pd_1: the number of depth1-classes in the target pattern p.
- pd_2: the number of depth2-classes which are related to a depth1-class in the target pattern p.

To analyze the complexity of the structural matching algorithm, we use a model of a source-class cluster which is presented in Figure 5.3. To simplify the problem, but without loss of generality, we assume that in the source-class cluster model there exists only one type of inter-class relation and each depth1 source-classes has d_2 depth2 source-classes.

The $Depth1Matching$ is conducted to find the source-class candidates for the pd_1 depth1-classes within the d_1 depth1 source-classes of the source-class cluster. Therefore, the running time is $\frac{d_1!}{(d_1-pd_1)!}$, i.e., $O(d_1^{pd_1})$.

For each ordered combination of depth1-classes, the $Depth2Matching$ is invoked to find the source-class candidates for all the depth2-classes. More specifically, for each depth1-class, we search to find the candidates for the pd_2 depth2-classes, which are related to this depth1-class, within d_2 depth2 source-classes of the source-class cluster. Since there exist d_1 depth1-classes, the total running time complexity is $d_1 \frac{d_2!}{(d_2-pd_2)!}$, i.e., $O(d_1 \cdot d_2^{pd_2})$.

As a result, the total running time complexity of the structural matching algorithms is $\frac{d_1!}{(d_1-pd_1)!} \cdot d_1 \cdot \frac{d_2!}{(d_2-pd_2)!}$, i.e., $O(d_1^{pd_1+1} \cdot d_2^{pd_2})$.

5.6 An Example

In this section, we will illustrate the operations of the two-phase design pattern detection process with a simple example. As shown in Figure 5.5, the search space $\mathcal{SP}_1$ of our example consists of classes $\{c_1, c_2, ..., c_{32}\}$. We take Bridge design pattern as our target design pattern whose class diagram and PDL representation are presented in Figure 5.4.

According to the proposed two-phase design pattern detection process, we start with parsing the PDL representation of the Bridge design pattern to obtain the attribute vector for the main-seed class $Implementor$. The main-seed class $Implementor$ is an abstract class and has two $Inherited\_By$ relations, one $in\_Association$ relation, therefore its attribute vector $Attr\_Vec(Implementor)$ is $[0, 2, 1, 0, 1]$.

In the approximate matching phase, the similarity function sim_{apx} is in-

Implementor (main-seed class)	Abstraction	Concrete-ImplementorA	Concrete-ImplementorB
c_2	c_1	c_{11}	c_{12}
c_2	c_1	c_{12}	c_{11}
c_3	c_2	c_9	c_{10}
c_3	c_2	c_{10}	c_9

Table 5.1: Combinations of matched source classes of depth1-classes

Implementor (main-seed class)	Abstraction	Concrete-ImplementorA	Concrete-ImplementorB	Refined-Abstraction-
c_3	c_2	c_9	c_{10}	c_{11}
c_3	c_2	c_9	c_{10}	c_{12}

Table 5.2: Identified instances of Bridge design pattern base on one combination

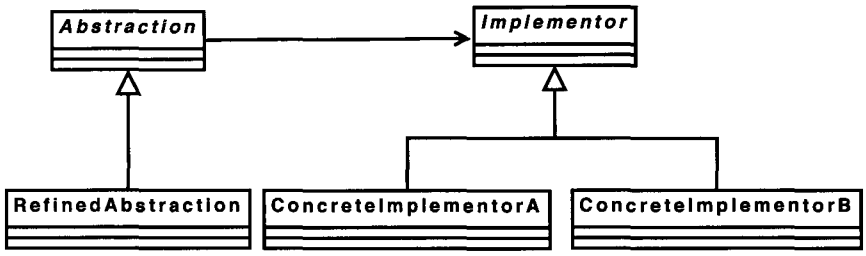
voked to compute the similarity value between the main-seed class *Implementor* and each source-class c_i in the search space $\mathcal{SP}_1$. According the obtained similarity values, we obtain two candidates c_2 and c_3 of the main-seed class *Implementor*, where the attribute vectors of these two candidates are both $[1, 2, 1, 1, 1]$. For these two candidates, we generate the corresponding source-class clusters cl_{c_2} and cl_{c_3} by adding all the related source-classes within two levels. The generated source-class cluster cl_{c_2} contains source-classes c_1, c_2, c_3, c_4, c_9, c_{10}, c_{11}, c_{12}, c_{16}, c_{18} and c_{30}, and the generated source-class cluster cl_{c_3} contains source-classes c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}, c_{17} and c_{18}, where c_2 and c_3 are the candidates of the main-seed class in each source-class cluster.

After finishing the approximate matching, we conduct the structural matching on the source-class clusters cl_{c_2} and cl_{c_3}. For each source-class cluster, we apply the functions $Depth1Matching()$ and $Depth2Matching()$ orderly. In the

function $Depth1Matching()$ we only collect the source-classes for the depth1-classes which are specified in the PDL representation of the Bridge pattern. Table 5.1 reports the result of executing the function $Depth1Matching()$.

In a further step, for each combination of matched depth1-classes we perform the $Depth2Matching()$ to identify complete instances of the Bridge design patten. To illustrate the process of $Depth2Matching()$, we take the combination $< c_3, c_2, c_9, c_{10} >$ in Table 5.1 as an example. First, we parse the PDL representation of the Bridge pattern to obtain all the inter-class relations between depth1-classes and depth2-classes, i.e., a list of triples < *seed-depth1-class, relation-type, depth2-class* >.

As shown in the PDL representation in Figure 5.4, Bridge pattern has only one depth2-class, $RefinedAbstraction$, thus the obtained triple is $<$ *Abstraction, Inherited_by, RefinedAbstraction* $>$. In the combination $< c_3, c_2, c_9, c_{10} >$ the corresponding source-class for *Abstraction* is c_2, then we search in the source-class cluster cl_{c_3} to find those classes by which c_2 is inherited. The results are c_{11} and c_{12}. Table 5.2 presents the identified design pattern instances based on the depth1-class combination $< c_3, c_2, c_9, c_{10} >$.




```
1   Begin-PDL
2     Pattern : Bridge
3       Main-seed class : Implementor
4         Depth1 :
5           Inherited_By :
6             ConcreteImplementorA;
7             ConcreteImplementorB
8           in_Association :
9               Abstraction
10        Depth2 :
11          Seed-Depth1 : Abstraction
12            Inherited_By :
13              RefinedAbstraction
14        AbstractClasses :
15          Implementor;
16          Abstraction
17    End-Pattern
18  End-PDL
```

Figure 5.4: Class diagram and PDL representation of Bridge design pattern.

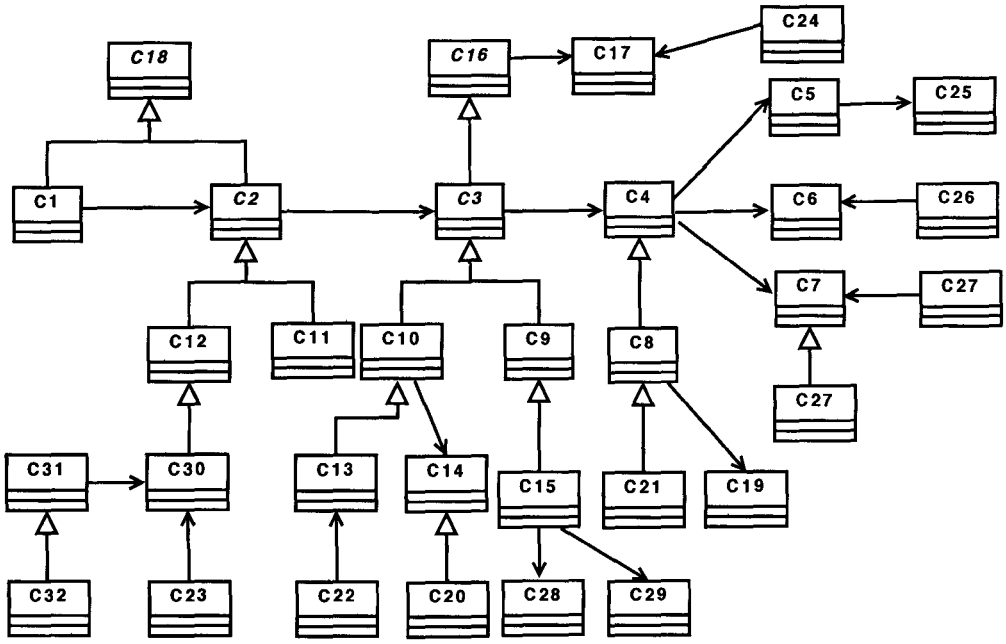


Figure 5.5: Class diagram of search space $\mathcal{SP}_1$.

Chapter 6

Case Study

In this chapter we apply the proposed feature-oriented design pattern detection approach on three versions of JHotDraw [3] systems. The case study is conducted in accordance with the proposed framework presented in Chapter 1. The process of the case study includes the following major steps: feature-specific scenario sets generation, execution pattern extraction, concept analysis and two-phase design pattern detection. The structure of this chapter is organized as follows. In Section 6.1, we give a description of our experimental hardware and software platform. In Section 6.2, we introduce the adopted subject systems used for our case study. Section 6.3 describes the process of applying the feature-oriented dynamic analysis on the subject systems. Section 6.4 presents the experimental results generated by applying the proposed two-phase design pattern detection approach. Finally, we discuss how we can exploit the obtained experimental results to support the task of migrating existing software systems into a software product line.

Systems	Version	# Classes	#Files	#LOC
JHotDraw	5.1	172	144	8419
JHotDraw	6.0b1	405	289	21091
JHotDraw	7.0.7	331	309	32122

Table 6.1: Statistics of the three versions JHotDraw systems.

6.1 Experimental Platform

The case study is performed on a Windows XP professional edition running on a laptop with a 1.5GHZ centrino processor, 1024M bytes memory and 1G bytes virtual memory. The implemented prototype toolkit is executed on Eclipse 3.2.1 [2], and as we mentioned in Chapter 4, we use Eclipse Test and Performance Tools Platform 4.2 as our profiling tool to collect the execution traces generated by running the scenarios in the feature-specific scenario set.

6.2 Subject System

We apply the proposed approach on a Java open-source project, *JHotDraw*, which is a Java framework for drawing two-dimensional graphics[3]. We select it for our case study due to the following reasons.

- JHotDraw was originally developed to demonstrate the good use of design patterns for designing software systems.
- The designers of JHotDraw indicate some applied design patterns in the documentation.
- JHotDraw is an open-source project with their source code available.

Table 6.1 presents several system statistics of the three versions of JHotDraw systems in the case study.

6.3 Feature-Oriented Dynamic Analysis

In the feature-orient dynamic analysis part, we aim to locate the core implementation classes for those features having high potential reusability and generate a mapping between features and feature-specific classes.

According to the proposed framework, this part of the case study consists of feature-specific scenario set generation, execution trace generation, execution pattern extraction and execution pattern analysis. In the remainder of this section, we will describe each stage in detail.

6.3.1 Feature-Specific Scenario Set Generation

First, by investigating the existing legacy documentation (e.g., user manuals, architectural descriptions and requirements documentation), we select the following ten key features which are of our interest as our target features for our case study.

- "Draw a Rectangle",
- "Draw a RoundRectangle"
- "Draw an Ellipse"
- "Draw a Polygon"
- "Draw a Line"
- "Draw a LineConnection"

- "Draw a Text"
- "Move a Figure"
- "Delete a Figure"
- "Group Figures"

In a further step, in order to extract the core implementation classes of each key feature, we devise a feature-specific scenario set for each subject feature. As an example, for the feature "Draw a Rectangle", we create a feature-specific scenario set below.

1 *start,* ***draw a rectangle****, exit*
2 *start,* ***draw a rectangle****, move, exit*
3 *start,* ***draw a rectangle****, edit, exit*
4 *start,* ***draw a rectangle****, delete, exit*

Next, for each feature, by running all the scenarios in its feature-specific scenario set on Eclipse Test and Performance Tools Platform (TPTP), we collect the corresponding execution traces and store them into an execution trace repository. In Figure 6.1, we list the average size of the execution traces of the scenarios of the 10 features in the three versions of JHotDraw systems.

6.3.2 Execution Pattern Extraction

In this stage, by applying a modified version of the sequential pattern mining algorithm [6] to the execution trace repository of each feature, we extract the execution patterns of each feature.

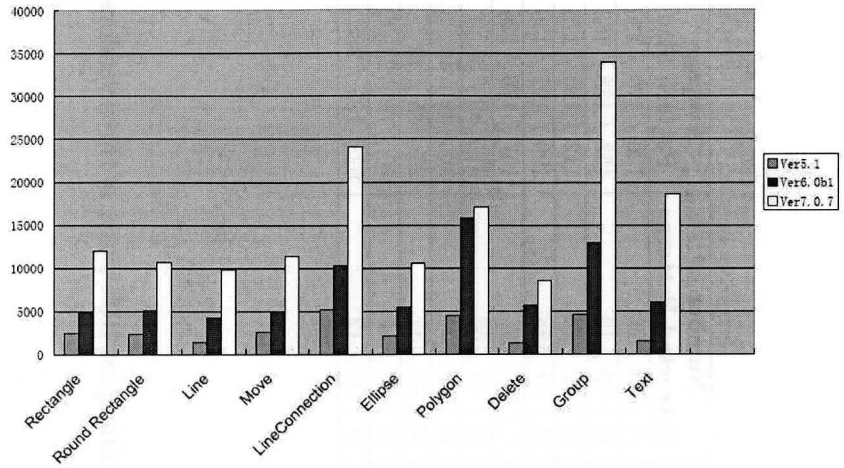


Figure 6.1: Average execution trace size of selected scenarios of the 10 features in three versions of JHotDraw systems.

Table 6.2 depicts the experimental results of execution trace extraction of the 10 features of the three versions of JHotDraw systems. For each feature the following statistics are provided:

- number of scenarios
- average trace size
- average pruned trace size
- number of extracted execution patterns
- average size of the execution patterns

Specific Feature of JHotDraw	Number of Scenarios	Average Trace Size	Average Pruned Trace Size	Number of Extracted Patterns	Average Pattern Size
Rectangle	4 / 4 / 4	2494 / 4889 / 11962	927 / 2165 / 2110	13 / 24 / 23	126 / 170 / 220
Round Rectangle	4 / 4 / 4	2369 / 5040 / 10620	927 / 2327 / 1864	15 / 25 / 19	153 / 138 / 183
Ellipse	4 / 4 / 4	2104 / 5492 / 10580	773 / 2226 / 1915	15 / 22 / 24	112 / 185 / 175
Polygon	4 / 4 / 4	4553 / 15769 / 17130	1654 / 4029 / 3142	21 / 41 / 38	199 / 192 / 130
Line	4 / 4 / 4	1439 / 4253 / 9882	546 / 2224 / 2123	7 / 24 / 27	157 / 170 / 126
Move	4 / 4 / 4	2599 / 4930 / 11341	774 / 2688 / 2487	18 / 34 / 52	31 / 89 / 37
Delete	4 / 4 / 4	1323 / 5739 / 8540	623 / 2456 / 969	16 / 32 / 24	36 / 89 / 49
Group	5 / 5 / 5	4579 / 12978 / 33921	1397 / 4675 / 4842	36 / 66 / 57	26 / 85 / 49
LineConnection	4 / 4 / 4	5238 / 10356 / 24075	1681 / 4158 / 4437	38 / 53 / 56	36 / 78 / 73
Text	4 / 4 / 4	1524 / 6074 / 18629	781 / 2435 / 2204	11 / 35 / 12	62 / 105 / 288

Legend: A / B / C
A: data for JHotDraw 5.1
B: data for JHotDraw 6.0b1
C: data for JHotDraw 7.0.7

Table 6.2: Results of execution trace extraction and execution pattern mining for 10 features of three versions of JHotDraw systems.

6.3.3 Execution Pattern Analysis

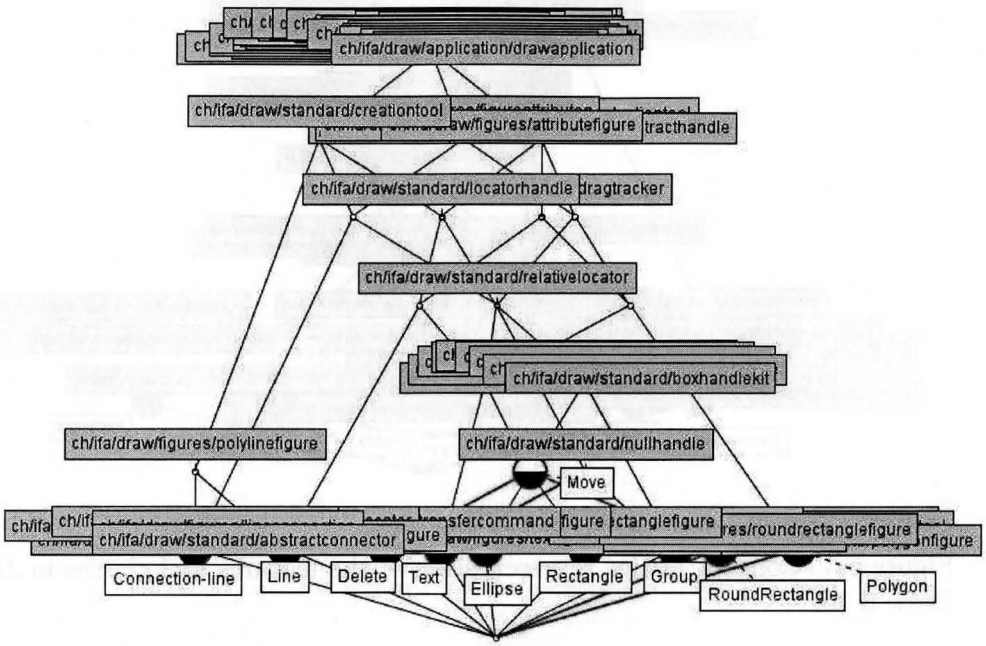


Figure 6.2: Concept lattice representation of the features and classes in JHotDraw 5.1.

In the execution pattern analysis stage, we employ concept analysis to produce a mapping between features and feature-specific classes. We supply the resulting execution patterns obtained from execution pattern extraction to a concept lattice generation tool, ConExp [5]. The generated concept lattices of JHotDraw 5.1, 6.0b1 and 7.0.7 are presented in Figures 6.2, 6.3 and 6.4 respectively, where each bubble represents a feature and the shaded labels represent classes. The feature-specific classes of each feature are gathered in the lower region, whereas the omnipresent classes are clustered in the upper region.

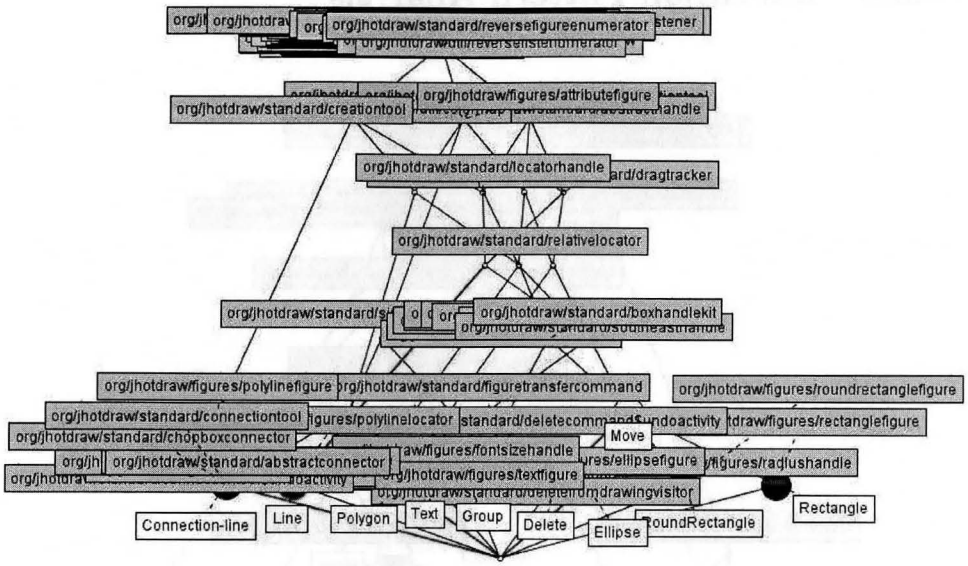


Figure 6.3: Concept lattice representation of the features and classes in JHotDraw 6.0b1.

Tables 6.3, 6.4 and 6.5 present a detailed description of the mapping between features and feature-specific classes of the three versions of JHotDraw systems.

6.4 Design Pattern Detection

In this part of the case study, for each JHotDraw system, we apply the proposed two-phase design pattern detection on the system to recover all the instances of the target design patterns. Then by analyzing the identified instances of the target design patterns and the mapping between features and feature-specific classes, we correlate the features to those identified design pattern instances.

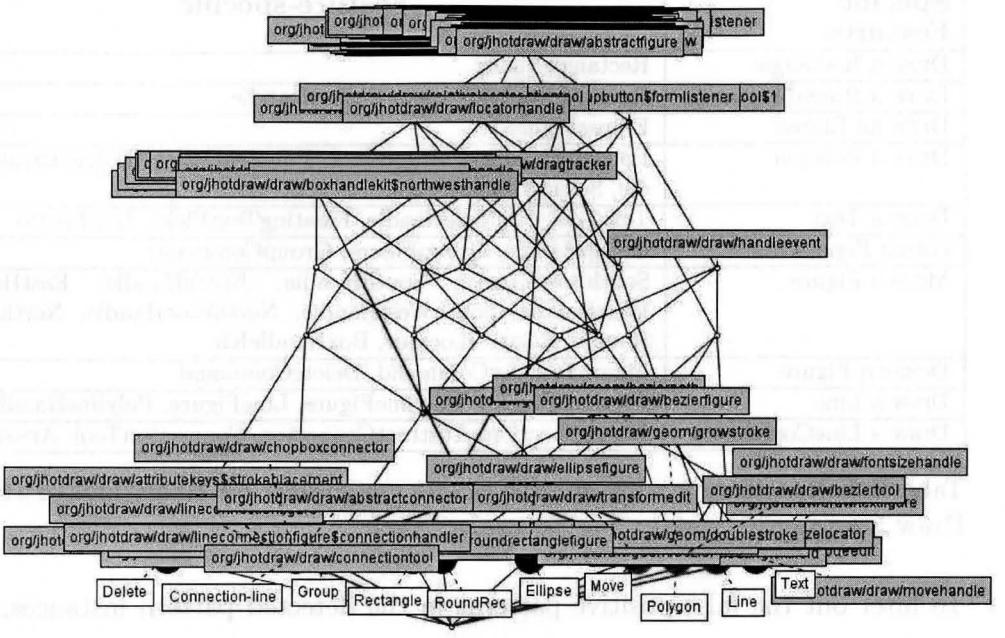


Figure 6.4: Concept lattice representation of the features and classes in JHotDraw 7.0.7.

6.4.1 Pattern Repository

Currently, our pattern repository contains the following design patterns: *Adapter, Proxy, Observer, Bridge* and *Strategy.* Each design pattern in the pattern repository is describe in the proposed PDL representation. As an example, Figure 6.5 illustrates the PDL representation of Bridge design pattern.

6.4.2 Results of Design Pattern Detection

For each subject system, we conduct the two-phase design pattern detection approach to identify each target design pattern in the design pattern repository.

Specific Features	Feature-specific Classes
Draw a Rectangle	RectangleFigure
Draw a RoundRectangle	RoundRectangleFigure, RadiusHandle
Draw an Ellipse	EllipseFigure
Draw a Polygon	PolygonHandle, PolygonTool, PolygonFigure, PolygonScaleHandle, SelectAreaTracker
Draw a Text	TextTool, FontSizeHandle, FloatingTextField, TextFigure
Group Figures	GroupHandle, GroupFigure, GroupCommand
Move a Figure	SouthEastHandle, SouthHandle, NorthHandle, EastHandle, WestHandle, SouthWestHandle, NorthEastHandle, NorthWestHandle, RelativeLocator, BoxHandleKit
Delete a Figure	FigureTransferCommand, DeleteCommand
Draw a Line	PolylineLocator, PolylineFigure, LineFigure, PolylineHandle
Draw a LineConnection	Lineconnection, AbstractConnector, ConnectionTool, ArrowTip

Table 6.3: Results of feature-specific classes assignment for 10 features of JHotDraw 5.1.

To filter out the false-positive patterns in the detected pattern instances, we perform a manual verification on these resulting pattern instances by inspecting the corresponding source code.

To correlate an identified design pattern instance with a feature, we check the overlap between the feature-specific classes of the feature (obtained from concept lattice) with the participating classes of the design pattern instance. If there exists an overlap, it means that there is a relation between the feature and the design pattern instance. For example, Figure 6.6 presents an instance of Strategy design pattern that is detected from JHotDraw 5.1. This pattern instance is related with the feature *Drawing a Polygon* since classes *PolygonHandle* and *PolygonScaleHandle* are feature-specific classes of the feature.

Table 6.6, 6.7, 6.8 and 6.9 present the correlation between the features and the identified design pattern instances in JHotDraw 5.1.

Specific Features	Feature-specific Classes
Draw a Rectangle	org/jhotdraw/figures/RectangleFigure
Draw a RoundRectangle	org/jhotdraw/figures/RoundRectangleFigure org/jhotdraw/figures/RadiusHandle
Draw an Ellipse	org/jhotdraw/figures/EllipseFigure
Draw a Polygon	org/jhotdraw/contrib/PolygonFigure org/jhotdraw/contrib/PolygonScaleHandle org/jhotdraw/contrib/PolygonHandle org/jhotdraw/contrib/PolygonTool
Draw a Line	org/jhotdraw/figures/PolylineFigure org/jhotdraw/figures/PolylineLocator org/jhotdraw/figures/LineFigure org/jhotdraw/figures/PolylineHandle
Move a Figure	org/jhotdraw/standard/SouthHandle org/jhotdraw/standard/NorthHandle org/jhotdraw/standard/EastHandle org/jhotdraw/standard/WestHandle org/jhotdraw/standard/SouthEastHandle org/jhotdraw/standard/SouthWestHandle org/jhotdraw/standard/NorthEastHandle org/jhotdraw/standard/NorthWestHandle org/jhotdraw/standard/RelativeLocator org/jhotdraw/standard/BoxHandleKit org/jhotdraw/standard/ResizeHandle
Delete a Figure	org/jhotdraw/standard/FigureTransferCommand org/jhotdraw/standard/DeleteCommand org/jhotdraw/standard/DeleteFromDrawingVisitor
Group Figures	org/jhotdraw/figures/GroupFigure org/jhotdraw/figures/GroupHandle org/jhotdraw/GroupCommand org/jhotdraw/standard/SelectAreaTracker
Draw a Text	org/jhotdraw/figures/TextFigure org/jhotdraw/figures/TextTool org/jhotdraw/figures/FontSizeHandle org/jhotdraw/util/FloatingTextField
Draw a LineConnection	org/jhotdraw/standard/AbstractConnector org/jhotdraw/figures/LineConnection org/jhotdraw/standard/ConnectionTool org/jhotdraw/figures/ArrowTip org/jhotdraw/figures/AbstractLineDecoration org/jhotdraw/standard/ChopBoxConnector

Table 6.4: Results of feature-specific classes assignment for 10 features of JHotDraw 6.0b1.

Table 6.10 and 6.11 present the correlation between the features and the identified design pattern instances in JHotDraw 6.0b1.

Table 6.12, 6.14, and 6.13 present the correlation between the features and the identified design pattern instances in JHotDraw 7.0.7.

6.5 Discussion

The results obtained from the two-phase pattern detection process can support the task of migrating existing software systems into a software product line. For example, Figure 6.7 presents three detected Adapter design pattern instances of feature *Draw a Polygon* in JHotDraw 5.1, 6.0b1 and 7.0.7, respectively. By comparing and analyzing the detected pattern instances of the three versions JHotDraw systems, we notice that the implementation of the feature *Draw a Polygon* in JHotDraw 5.1 and 6.0b1 is very similar, while there exist some differences between the implementation in JHotDraw 7.0.7 with that in JHotDraw 5.1 and 6.0b1. Performing such comparison and analysis on the detected pattern instances of these 10 features, which are used most frequently in the applications and shared by all the three versions of JHotDraw systems, can help to comprehend the features' implementation in a design level and allows for a quick understanding of evolution of the features within the software.

Specific Features	**Feature-specific Classes**
Draw a Rectangle	org/jhotdraw/draw/RectangleFigure
Draw a RoundRectangle	org/jhotdraw/draw/RoundRectangleFigure org/jhotdraw/draw/RoundRectRadiusHandle
Draw an Ellipse	org/jhotdraw/draw/EllipseFigure
Draw a Polygon	org/jhotdraw/draw/BezierTool org/jhotdraw/geom/GrowStroke org/jhotdraw/geom/DoubleStroke org/jhotdraw/draw/BezierScaleHandle org/jhotdraw/draw/BezierFigure org/jhotdraw/geom/Bezier
Draw a Line	org/jhotdraw/draw/LineFigure org/jhotdraw/draw/BezierNodeEdit org/jhotdraw/draw/BezierNodeHandle org/jhotdraw/draw/HandleTracker org/jhotdraw/draw/HandleMulticaster
Move a Figure	org/jhotdraw/draw/TransformEdit org/jhotdraw/draw/MoveHandle
Delete a Figure	org/jhotdraw/draw/AbstractDrawing org/jhotdraw/app/action/DeleteAction
Group Figures	org/jhotdraw/draw/AbstractCompositeFigure org/jhotdraw/draw/action/GroupAction
Draw a Text	org/jhotdraw/draw/TextFigure org/jhotdraw/draw/FontSizeHandle org/jhotdraw/draw/FontSizeLocator org/jhotdraw/draw/TextTool org/jhotdraw/geom/Insets2DDouble org/jhotdraw/draw/FloatingTextField
Draw a LineConnection	org/jhotdraw/draw/LineConnectionFigure org/jhotdraw/draw/AbstractConnector org/jhotdraw/draw/ChopBoxConnector org/jhotdraw/draw/ConnectionTool org/jhotdraw/draw/AttributeKey

Table 6.5: Results of feature-specific classes assignment for 10 features of JHotDraw 7.0.7.

```
1  Begin-PDL
2   Pattern : Bridge
3    Main-seed class : Implementor
4     Depth1 :
5      Inherited_By :
6       ConcreteImplementor
7      in_Association :
8       Abstraction
9     Depth2 :
10      Seed-Depth1 : Abstraction
11       Inherited_By :
12          RefinedAbstraction
13    AbstractClasses :
14      Implementor;
15      Abstraction
16   End-Pattern
17  End-PDL
```

Figure 6.5: Class diagram and PDL representation of Bridge design pattern.

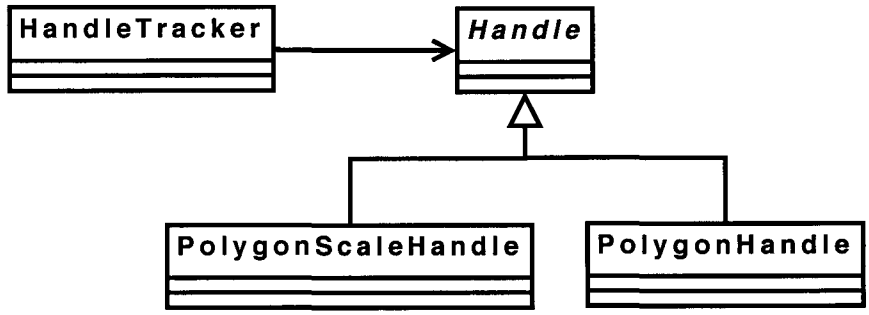


Figure 6.6: A Strategy design pattern instance of feature *Drawing a Polygon* in JHotDraw 5.1.

Design Pattern Instance	Related Feature
CH/ifa/draw/figures/PolyLineFigure (Target) CH/ifa/draw/figures/LineConnection (Adapter) CH/ifa/draw/framework/Connector (Adaptee)	Draw a Line
CH/ifa/draw/standard/AbstractHandle (Target) CH/ifa/draw/figures/RadiusHandle (Adapter) CH/ifa/draw/figures/RoundRectangleFigure (Adaptee)	Draw a RoundRectangle
CH/ifa/draw/standard/AbstractHandle (Target) CH/ifa/draw/contrib/PolygonHandle (Adapter) CH/ifa/draw/framework/Locator (Adaptee)	Draw a Polygon
CH/ifa/draw/standard/AbstractTool (Target) CH/ifa/draw/contrib/PolygonTool (Adapter) CH/ifa/draw/contrib/PolygonFigure (Adaptee)	Draw a Polygon
CH/ifa/draw/util/Command (Target) CH/ifa/draw/figures/GroupCommand (Adapter) CH/ifa/draw/framework/DrawingView (Adaptee)	Group Figures
CH/ifa/draw/framework/Connector (Target) CH/ifa/draw/standard/AbstractConnector (Adapter) CH/ifa/draw/framework/Figure (Adaptee)	Draw a LineConnection
CH/ifa/draw/framework/ConnectionFigure (Target) CH/ifa/draw/figures/LineConnection (Adapter) CH/ifa/draw/framework/Connector (Adaptee)	Draw a LineConnection
CH/ifa/draw/figures/PolyLineFigure (Target) CH/ifa/draw/figures/LineConnection (Adapter) CH/ifa/draw/framework/Connector (Adaptee)	Draw a LineConnection

Table 6.6: Results of identified Adapter design pattern instances and related features in JHotDraw 5.1 system

Design Pattern Instance	Related Feature
CH/ifa/draw/figures/PolyLineFigure (Abstraction) CH/ifa/draw/figures/LineConnection (RefinedAbstraction) CH/ifa/draw/figures/LineDecoration (Implementor) CH/ifa/draw/figures/ArrowTip (ConcreteImplementor)	Draw a LineConnection
CH/ifa/draw/figures/LineConnection (Abstraction) CH/ifa/draw/figures/ElbowConnection (RefinedAbstraction) CH/ifa/draw/framework/Connector (Implementor) CH/ifa/draw/standard/AbstractConnector (ConcreteImplementor)	Draw a LineConnection

Table 6.7: Results of identified Bridge design pattern instances and related features in JHotDraw 5.1 system

Design Pattern Instance	Related Feature
CH/ifa/draw/framework/Figure (Observer) CH/ifa/draw/framework/ConnectionFigure (ConcreteObserver) CH/ifa/draw/standard/CompositeFigure (Subject) CH/ifa/draw/figures/GroupFigure (ConcreteSubject)	Draw a LineConnection
CH/ifa/draw/framework/Figure (Observer) CH/ifa/draw/framework/ConnectionFigure (ConcreteObserver) CH/ifa/draw/standard/CompositeFigure (Subject) CH/ifa/draw/standard/StandardDrawing (ConcreteSubject)	Draw a LineConnection

Table 6.8: Results of identified Observer design pattern instances and related features in JHotDraw 5.1 system

Design Pattern Instance	Related Feature
CH/ifa/draw/figures/PolyLineFigure (Context) CH/ifa/draw/figures/LineDecoration (Strategy) CH/ifa/draw/figures/ArrowTip (ConcreteStrategy)	Draw a Line
CH/ifa/draw/figures/PolyLineFigure (Context) CH/ifa/draw/figures/LineDecoration (Strategy) CH/ifa/draw/figures/ArrowTip (ConcreteStrategy)	Draw a LineConnection
CH/ifa/draw/standard/FigureTransferCommand (Context) CH/ifa/draw/framework/DrawingView (Strategy) CH/ifa/draw/standard/StandardDrawingView (ConcreteStrategy)	Delete a Figure
CH/ifa/draw/figures/TextTool (Context) CH/ifa/draw/standard/TextHolder (Strategy) CH/ifa/draw/figures/TextFigure (ConcreteStrategy)	Draw a Text
CH/ifa/draw/standard/AbstractConnector (Context) CH/ifa/draw/framework/Figure (Strategy) CH/ifa/draw/framework/ConnectionFigure (ConcreteStrategy)	Draw a LineConnection
CH/ifa/draw/standard/AbstractConnector (Context) CH/ifa/draw/framework/Figure (Strategy) CH/ifa/draw/framework/ConnectionFigure (ConcreteStrategy)	Draw a LineConnection
CH/ifa/draw/figures/LineConnection (Context) CH/ifa/draw/framework/Connector (Strategy) CH/ifa/draw/standard/AbstractConnector (ConcreteStrategy)	Draw a LineConnection
CH/ifa/draw/contrib/PolygonHandle (Context) CH/ifa/draw/standard/AbstractLocator (Strategy) CH/ifa/draw/standard/AbstractConnector (ConcreteStrategy)	Draw a Polygon

Table 6.9: Results of identified Strategy design pattern instances and related features in JHotDraw 5.1 system

Design Pattern Instance	Related Feature
org/jhotdraw/framework/FigureVisitor (Target) org/jhotdraw/standard/DeleteFromDrawingVisitor (Adapter) org/jhotdraw/framework/Drawing (Adaptee)	Delete a Figure
org/jhotdraw/standard/AbstractHandle (Target) org/jhotdraw/figures/RadiusHandle (Adapter) org/jhotdraw/figures/RoundRectangleFigure (Adaptee)	Draw a RoundRectangle
org/jhotdraw/framework/Figure (Target) org/jhotdraw/figures/TextFigure (Adapter) org/jhotdraw/standard/OffsetLocator (Adaptee)	Draw a Text
org/jhotdraw/standard/AbstractTool (Target) org/jhotdraw/contrib/PolygonTool (Adapter) org/jhotdraw/contrib/PolygonFigure (Adaptee)	Draw a Polygon
org/jhotdraw/framework/Connector (Target) org/jhotdraw/standard/AbstractConnector (Adapter) org/jhotdraw/framework/Figure (Adaptee)	Draw a LineConnection
org/jhotdraw/framework/Figure (Target) org/jhotdraw/figures/LineConnection (Adapter) org/jhotdraw/framework/Connector (Adaptee)	Draw a LineConnection

Table 6.10: Results of identified Adapter design pattern instances and related features in JHotDraw 6.0b1 system

Design Pattern Instance	Related Feature
org/jhotdraw/standard/DeleteFromDrawingVisitor (Context) org/jhotdraw/framework/Drawing (Strategy) org/jhotdraw/standard/StandardDrawing (ConcreteStrategy)	Delete a Figure
org/jhotdraw/util/UndoableCommand (Context) org/jhotdraw/util/Command (Strategy) org/jhotdraw/standard/Deletecommand (ConcreteStrategy)	Delete a Figure
org/jhotdraw/figures/TextTool (Context) org/jhotdraw/standard/TextHolder (Strategy) org/jhotdraw/figures/TextFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/figures/TextAreaTool (Context) org/jhotdraw/standard/TextHolder (Strategy) org/jhotdraw/figures/TextFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/contrib/PolygonHandle (Context) org/jhotdraw/framework/Locator (Strategy) org/jhotdraw/figures/PolyLineLocator (ConcreteStrategy)	Draw a Polygon
org/jhotdraw/contrib/PolygonHandle (Context) org/jhotdraw/framework/Locator (Strategy) org/jhotdraw/standard/OffsetLocator (ConcreteStrategy)	Draw a Polygon
org/jhotdraw/figures/PolyLineFigure (Context) org/jhotdraw/figures/LineDecoration (Strategy) org/jhotdraw/figures/ArrowTip (ConcreteStrategy)	Draw a Line
org/jhotdraw/standard/AbstractConnector (Context) org/jhotdraw/framework/Figure (Strategy) org/jhotdraw/figures/LineConnection (ConcreteStrategy)	Draw a LineConnection
org/jhotdraw/figures/LineConnection (Context) org/jhotdraw/framework/Connector (Strategy) org/jhotdraw/standard Chopboxconnector (ConcreteStrategy)	Draw a LineConnection
org/jhotdraw/standard/ConnectionTool (Context) org/jhotdraw/framework/ConnectionFigure (Strategy) org/jhotdraw/standard/Chopboxconnector (ConcreteStrategy)	Draw a LineConnection

Table 6.11: Results of identified Strategy design pattern instances and related features in JHotDraw 6.0b1 system

Design Pattern Instance	Related Feature
org/jhotdraw/draw/AbstractTool (Target) org/jhotdraw/draw/BezierTool (Adapter) org/jhotdraw/draw/BezierFigure (Adaptee)	Draw a Polygon
org/jhotdraw/draw/ConnectionFigure (Target) org/jhotdraw/draw/LineConnectionFigure (Adapter) org/jhotdraw/draw/Connector (Adaptee)	Draw a LineConnection

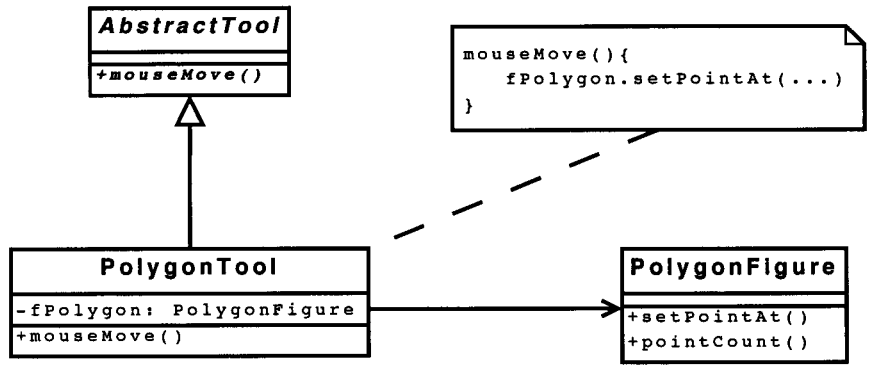
Table 6.12: Results of identified Adapter design pattern instances and related features in JHotDraw 7.0.7 system

Design Pattern Instance	Related Feature
org/jhotdraw/draw/action/GroupAction (Context) org/jhotdraw/draw/CompositeFigure (Strategy) org/jhotdraw/draw/GraphicalCompositeFigure (ConcreteStrategy)	Group Figures
org/jhotdraw/draw/TextTool (Context) org/jhotdraw/draw/TextHolder (Strategy) org/jhotdraw/draw/TextFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/draw/TextTool (Context) org/jhotdraw/draw/TextHolder (Strategy) org/jhotdraw/draw/TextAreaFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/draw/TextAreaTool (Context) org/jhotdraw/draw/TextHolder (Strategy) org/jhotdraw/draw/TextAreaFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/draw/TextAreaTool (Context) org/jhotdraw/draw/TextHolder (Strategy) org/jhotdraw/draw/TextFigure (ConcreteStrategy)	Draw a Text
org/jhotdraw/draw/LineConnectionFigure (Context) org/jhotdraw/draw/Connector (Strategy) org/jhotdraw/draw/ChopBoxConnector (ConcreteStrategy)	Draw a LineConnection

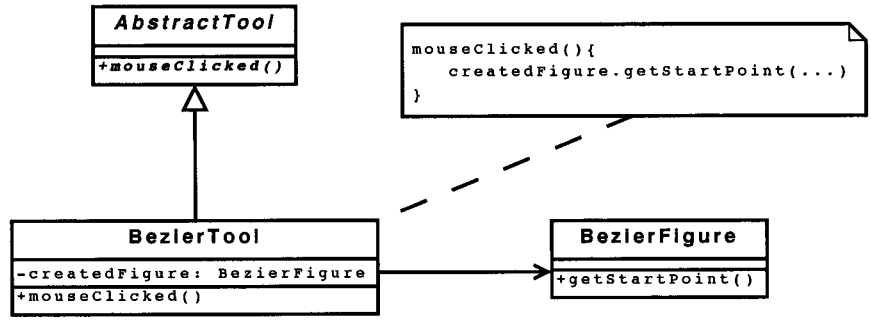
Table 6.13: Results of identified Strategy design pattern instances and related features in JHotDraw 7.0.7 system

Design Pattern Instance	Related Feature
org/jhotdraw/draw/AbstractConnector (Abstraction) org/jhotdraw/draw/ChopBoxConnector (RefinedAbstraction) org/jhotdraw/draw/Figure (Implementor) org/jhotdraw/draw/LineConnectionFigure (ConcreteImplementor)	Draw a LineConnection

Table 6.14: Results of identified Bridge design pattern instances and related features in JHotDraw 7.0.7 system



a. JHotDraw 5.1 and 6.0b1



b. JHotDraw 7.0.7

Figure 6.7: Adapter design pattern instances related to feature Drawing a Polygon in JHotDraw 5.1, 6.0b1 and 7.0.7.

Chapter 7

Conclusion and Future Work

In this thesis we presented a method to identify individual design pattern instances from the implementation of system behavioral features. The main advantage of our approach over the existing design pattern recovery approaches is incorporating dynamic feature analysis into design recovery. This allows us to perform a goal-driven design pattern detection and focus ourselves on design patterns that implement specific software functionality as opposed to conducting a general pattern detection which are susceptible to high complexity problem.

The major parts of the proposed approach are summarized as follows. The first part (feature-oriented dynamic analysis) consists of feature and scenario identification, execution pattern mining, and concept analysis to produce a mapping between features and feature-specific classes, which is used to correlate the features to the identified design pattern instances. In the second part (two-phase design pattern detection), a target design pattern is represented in a novel design pattern description language, PDL, which uses a center-role main-seed class, depth1-classes, depth2-classes and the inter-class relations among them

to describe the structural information of the design pattern. The matching process consists of an approximate matching and a structural matching that identify the specified pattern while providing scalability of the process.

We have applied our proposed approach on three versions of JHotDraw systems and obtained a very promising experimental results in both feature analysis and design pattern detection. Finally, we have implemented a prototype toolkit for the proposed approach on the Eclipse open platform.

7.1 Discussion

In Table 7.1 we compare our technique with several current major design pattern detection techniques based on different criteria such as: category of techniques; kind of pattern description language; automatic or human assisted; and applicable programming language.

Design Pattern Detection Techniques	Category of Techniques	Pattern Description Language	Applicable Prog-ramming Language	Degree of human involvement
Jing D. [17]	matix-based	NA	Java	automatic
Nija Shi [37]	structure-based	NA	Java	automatic
Zsolt Balanyi [11]	structure-based	DPML	C++	automatic
Nikolaos [43]	matix-based	NA	Java	automatic
Lucia [31]	structure-based	VL	Java	automatic
Antonial [7]	metric-based	AOL	C++	automatic
Yann-Gael [23]	metric-based	NA	Java	automatic
Vassilios [45]	structure-based	RSF & REQL	Effiel	semi-automatic
C. Kramer [29]	structure-based	PROLOG	SmallTalk	automatic
Our technique	**feature-oriented**	**PDL**	**Java**	**semi-automatic**

Table 7.1: Comparison of several design pattern detection techniques

7.2 Future Work

Our future work will mainly concentrate on the following directions:

- extracting more inter-class relations, such as delegation and method invocation, to improve the accuracy of the result of design pattern detection process.
- applying our proposed approach on a real evolutionary development of software product line.
- tracking the evolution of software systems at design level by analyzing the evolution of design patterns.
- building a benchmark which allow us to compare different design pattern detection systems and evaluate the results of the different approaches.

Bibliography

[1] Asm home page. http://asm.objectweb.org/, 2006.

[2] Eclipse version 3.2. http://www.eclipse.org.

[3] Jhotdraw start page. http://www.jhotdraw.org, 2006.

[4] The eclipse test and performance tools platform, 2006. http://www.eclipse.org/tptp.

[5] Formal concept analysis toolkit version 1.0.1. http://sourceforge.net/projects/conexp.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[7] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 153, Washington, DC, USA, 1998. IEEE Computer Society.

[8] G. Antoniol and Y. Gueheneuc. Feature identification: A novel approach and a case study. In *ICSM '05: Proceedings of the 21st IEEE Interna-*

tional Conference on Software Maintenance (ICSM'05), pages 357–366, Washington, DC, USA, 2005. IEEE Computer Society.

[9] G. Arevalo, F. Buchli, and O. Nierstrasz. Detecting implicit collaboration patterns. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 122–131, Washington, DC, USA, 2004. IEEE Computer Society.

[10] G. Arevalo and T. Mens. Analysing object-oriented application frameworks using concept analysis. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 53–63, London, UK, 2002. Springer-Verlag.

[11] Z. Balanyi and R. Ferenc. Mining design patterns from c++ source code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 305, Washington, DC, USA, 2003. IEEE Computer Society.

[12] J. Bayer, J. Girard, M. Würthner, J. DeBaud, and M. Apel. Transitioning legacy assets to a product line architecture. *SIGSOFT Softw. Eng. Notes*, 24(6):446–463, 1999.

[13] G. Birkhoff. *Lattice Theory.* American Mathematical Society, 1940.

[14] K. Brown. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical report, Raleigh, NC, USA, 1996.

[15] P. Clements and L. Northrop. A framework for software product line practice. Technical report, www.sei.cmu.edu/productlines/framework.html, 2004.

[16] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Case studies of visual language based design pattern recovery. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 163–172, Bari, Italy, 2006. IEEE CS Press.

[17] J. Dong, D. S. Lad, and Y. Zhao. Dp-miner: Design pattern discovery using matrix. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 371–380, Washington, DC, USA, 2007. IEEE Computer Society.

[18] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 602–611, Washington, DC, USA, 2001. IEEE Computer Society.

[19] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29:210 – 224, March 2003.

[20] E. Gamma, R.Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[21] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. Translator-C. Franzke.

[22] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 314–323, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Y. Gueheneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 172–181, Washington, DC, USA, 2004. IEEE Computer Society.

[24] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.

[25] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 159, Washington, DC, USA, 2002. IEEE Computer Society.

[26] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of the 1st ICSE International Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, 2003.

[27] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st in-*

ternational conference on Software engineering, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[28] H. Kim and C. Boldyreff. A method to recover design patterns using software product metrics. In *ICSR-6: Proceedings of the 6th International Conerence on Software Reuse*, pages 318–335, London, UK, 2000. Springer-Verlag.

[29] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering*, pages 208–215, Washington, DC, USA, 1996. IEEE Computer Society.

[30] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 349–359, New York, NY, USA, 1997. ACM Press.

[31] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. A two phase approach to design pattern recovery. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society.

[32] R.Ferenc, A. Beszedes, M. Tarkiainen, and T. Gyimothy. Columbus - reverse engineering tool and schema for c++. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 172, Washington, DC, USA, 2002. IEEE Computer Society.

[33] H. Safyallah. Dynamic analysis of software systems based on sequential pattern mining. Master's thesis, Department of Computing and Software, Mcmaster University, 2006.

[34] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 84–88, Washington, DC, USA, 2006. IEEE Computer Society.

[35] K. Sartipi. *Software Architecture Recovery based on Pattern Matching.* PhD thesis, School of Computer Science, University of Waterloo, 2003.

[36] K. Sartipi and H. Safyallah. Application of execution pattern mining and concept lattice analysis on software structure evaluation. In *SEKE*, pages 302–308, 2006.

[37] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[38] M. Siff and T. W. Reps. Identifying modules via concept analysis. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 170–179, Washington, DC, USA, 1997. IEEE Computer Society.

[39] G. Snelting. Software reengineering based on concept lattices. In *CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[40] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–110, New York, NY, USA, 1998. ACM Press.

[41] T. Systa. Dynamic reverse engineering of java software. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 174–175, London, UK, 1999. Springer-Verlag.

[42] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 230, Washington, DC, USA, 1999. IEEE Computer Society.

[43] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006.

[44] M. Vokac. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5:139–157, 2006.

[45] W. Wang and V. Tzerpos. Design pattern detection in eiffel systems. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 165–174, Washington, DC, USA, 2005. IEEE Computer Society.

[46] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Trans. Softw. Eng.*, 18(12):1038–1044, 1992.

[47] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 134–142, Washington, DC, USA, 2005. IEEE Computer Society.