A dissertation submitted to the

FAKULTÄT FÜR INFORMATIK UND AUTOMATISIERUNG

TECHNISCHE UNIVERSITÄT ILMENAU

Customizable Feature based Design Pattern Recognition Integrating Multiple Techniques

for the degree of

DOKTOR-INGENIEUR (DR.-ING.)

presented by

Ghulam Rasool (MSCS)

born April 8, 1975

in Pakpattan, Pakistan

Gutachter:

1. **Prof. Dr.-Ing. habil. Ilka Philippow**
2. **Prof. Dr.-Ing. habil. Kai-Uwe Sattler**
3. **Dr.-Ing. Patrick Mäder**

October 2010

urn:nbn:de:gbv:ilm1-2011000079

Abstract

Recovering design information from legacy applications is a complex, expensive, quiet challenging, and time consuming task due to ever increasing complexity of software and advent of modern technology. The growing demand for maintenance of legacy systems, which can cope with the latest technologies and new business requirements, the reuse of artifacts from the existing legacy applications for new developments become very important and vital for software industry. Due to constant evolution in architecture of legacy systems, they often have incomplete, inconsistent and obsolete documents which do not provide enough information about the structure of these systems. Mostly, source code is the only reliable source of information for recovering artifacts from legacy systems. Extraction of design artifacts from the source code of existing legacy systems supports program comprehension, maintenance, code refactoring, reverse engineering, redocumentation and reengineering methodologies.

The objective of approach used in this thesis is to recover design information from legacy code with particular focus on the recovery of design patterns. Design patterns are key artifacts for recovering design decisions from the legacy source code. Patterns have been extensively tested in different applications and reusing them yield quality software with reduced cost and time frame. Different techniques, methodologies and tools are used to recover patterns from legacy applications in the past. Each technique recovers patterns with different precision and recall rates due to different specifications and implementations of same pattern. The approach used in this thesis is based on customizable and reusable feature types which use static and dynamic parameters to define variant pattern definitions. Each feature type allows user to switch/select between multiple searching techniques (SQL queries, Regular Expressions and Source Code Parsers) which are used to match features of patterns with source code artifacts. The technique focuses on detecting variants of different design patterns by using static, dynamic and semantic analysis techniques. The integrated use of SQL queries, source code parsers, regular expressions and annotations improve the precision and recall for pattern extraction from different legacy systems. The approach has introduced new semantics of annotations to be used in the source code of legacy applications, which reduce search space and time for detecting patterns.

The prototypical implementation of approach, called UDDPRT is used to recognize different design patterns from the source code of multiple languages (Java, C/C++, C#).

The prototype is flexible and customizable that novice user can change the SQL queries and regular expressions for detecting implementation variants of design patterns. The approach has improved significant precision and recall of pattern extraction by performing experiments on number of open source systems taken as baselines for comparisons.

Zusammenfassung

Die Analyse und Rückgewinnung von Architekturinformationen aus existierenden Altsystemen ist eine komplexe, teure und zeitraubende Aufgabe, was der kontinuierlich steigenden Komplexität von Software und dem Aufkommen der modernen Technologien geschuldet ist. Die Wartung von Altsystemen wird immer stärker nachgefragt und muss dabei mit den neuesten Technologien und neuen Kundenanforderungen umgehen können. Die Wiederverwendung der Artefakte aus Altsystemen für neue Entwicklungen wird sehr bedeutsam und überlebenswichtig für die Softwarebranche. Die Architekturen von Altsystemen unterliegen konstanten Veränderungen, deren Projektdokumentation oft unvollständig, inkonsistent und veraltet ist. Diese Dokumente enthalten ungenügend Informationen über die innere Struktur der Systeme.

Häufig liefert nur der Quellcode zuverlässige Informationen über die Struktur von Altsystemen. Das Extrahieren von Artefakten aus Quellcode von Altsystemen unterstützt das Programmverständnis, die Wartung, das Refactoring, das Reverse Engineering, die nachträgliche Dokumentation und Reengineering Methoden. Das Ziel dieser Dissertation ist es Entwurfsinformationen von Altsystemen zu extrahieren, mit Fokus auf die Wiedergewinnung von Architekturmustern. Architekturmuster sind Schlüsselelemente, um Architekturentscheidungen aus Quellcode von Altsystemen zu extrahieren. Die Verwendung von Mustern bei der Entwicklung von Applikationen wird allgemein als qualitätssteigernd betrachtet und reduziert Entwicklungszeit und kosten. In der Vergangenheit wurden unterschiedliche Methoden entwickelt, um Muster in Altsystemen zu erkennen. Diese Techniken erkennen Muster mit unterschiedlicher Genauigkeit, da ein und dasselbe Muster unterschiedlich spezifiziert und implementiert wird. Der Lösungsansatz dieser Dissertation basiert auf anpassbaren und wiederverwendbaren Merkmal-Typen, die statische und dynamische Parameter nutzen, um variable Muster zu definieren. Jeder Merkmal-Typ verwendet eine wählbare Suchtechnik (SQL Anfragen, Reguläre Ausdrücke oder Quellcode Parser), um ein bestimmtes Merkmal eines Musters im Quellcode zu identifizieren. Insbesondere zur Erkennung verschiedener Varianten eines Musters kommen im entwickelten Verfahren statische, dynamische und semantische Analysen zum Einsatz. Die Verwendung unterschiedlicher Suchtechniken erhöht die Genauigkeit der Mustererkennung bei verschiedenen Softwaresystemen. Zusätzlich wurde eine neue Semantik für Annotationen im Quellcode von existierenden Softwaresystemen entwickelt, welche die Effizienz der Mustererkennung steigert.

Eine prototypische Implementierung des Ansatzes, genannt UDDPRT, wurde zur Erkennung verschiedener Muster in Softwaresystemenen unterschiedlicher Programmiersprachen (JAVA, C/C++, C#) verwendet. UDDPRT erlaubt die Anpassung der Mustererkennung durch den Benutzer. Alle Abfragen und deren Zusammenspiel sind konfigurierbar und erlauben dadurch die Erkennung von neuen und abgewandelten Mustern. Es wurden umfangreiche Experimente mit diversen Open Source Software Systemen durchgeführt und die erzielten Ergebnisse wurden mit denen anderer Ansätze verglichen. Dabei war es möglich eine deutliche Steigerung der Genauigkeit im entwickelten Verfahren gegenüber existierenden Ansätzen zu zeigen.

Acknowledgements

First and far most, this work would not have been completed except by special blessings of Almighty Allah. There are several other individuals who made contributions to this thesis. Tireless efforts, expert guidance and analytic thinking on the part of my supervisor, Professor Dr. Ilka Philippow, who spent countless hours by discussing different aspects of my thesis. She always kept door open in her tight time schedule and dedicated countless hours for discussions. Her valuable suggestions and continuous encouragement helped during difficult time of my dissertation. I greatly appreciate her immeasurable efforts which exposed my scientific research skills and sparked my interest in the area of design pattern recovery.

My debit of gratitude goes to Dr. Patrick Mäder who helped in different aspects of this thesis. His friendly attitude and scientific support was admirable throughout difficult journey of my PhD. He polished my raw theoretical ideas, reviewed my papers very carefully and provided valuable feedback. I cannot forget his help to solve many technical issues related with my thesis and his moral support played a great role for the completion of this work. I would also like to thank Professor Dr. Nazir A. Zafar, Dean Faculty of Information Technology, University of Central Punjab, Lahore, Pakistan, who provided research environment for working during my stay in Pakistan.

I would also like to give my gratitude to the students who supported me in the implementation of prototype. My thanks go to Sebastian Kahlm and Xiang Yu for their dedicated work.

I would like to thank the Higher Education Commission (HEC) and the Govt of Pakistan for funding my PhD study in Germany for the period of three years.

My sincere thanks go to Nils Würfel and Heiner Kotula for their technical support during my stay in TU Ilmenau. I would also like to thank my colleague Elke Bouillon for her help in day to day official matters.

Apart from study time, it was quite challenging at beginning to spend free time in a small city like Ilmenau without good friends. I feel lucky to have very good friends who helped me in all aspects of life. My thanks go to Muhammad Nasir, Abdil Basit, Rasheed Sadiq and Arfan Mansoor for their continuous support and company in difficult times of my thesis. My special thanks go to Ramzan Talib, who shared his day to day experience about research and non-research issues through long telephonic discussions during late night hours.

Finally, I have to appreciate the irreplaceable support of my wife, who has given me awareness to polish my skills and provide me with invaluable support for my research. She supported me in this challenging time and took care of children alone in my absence. Thanks Suriya.

Dedication

To my parents, family and children

Contents

Chapter 1

Introduction

1.1 Overview

Program comprehension techniques are assisted by different tools which extract different artifacts from source code such as class models, inter-class relationships, execution traces, call graphs, etc. Some of the tools are also able to extract different UML diagrams from source code of object oriented programming languages. These diagrams do not convey hidden intensions of developers existing in the source code. The understanding of source code without documentation and other clues is extremely difficult task. Mostly, documents associated with source code are not available or they are obsolete, inconsistent and vague. The only reliable source for extracting information from legacy applications is the source code. Statistics reflect that fifty to ninety percent of the time, depending upon the system under consideration is spent on program understanding [14]. According to IBM survey report of different legacy applications, 250 billion lines of source code are maintained in 2000 [15]. It is also reported in another study that old languages are still not dead and 80% of IT systems are running on legacy platforms [16]. Similarly, Caper Jones study [4] shows that companies spent 40% of their time on software maintenance. Most existing tools are supporting object oriented programming languages, but they lack similar support for procedural languages which really require restructuring, refactoring and reengineering due to the evolution of technology and new business requirements. The maintenance cost of software is increasing with respect to time and the use of reverse engineering is gaining more and more attention in the field of legacy and embedded applications.

The program comprehension tools should be capable to extract the intent and design of the source code. Design recovery is the key to program comprehension. Design recovery should produce the information required to understand what a program does, how it does it and why it does it. In this context, design patterns are the key artifacts because of their unique design rationale. Design patterns are very important artifacts to extract design information from the legacy applications. Recovering the design pattern instances is important for program comprehension, maintenance, restructuring, refactoring, reverse engineering and reengineering of existing legacy applications. Thus, by extracting design patterns from source code, it is possible to reveal the intent and design of the software

system. We believe that by tracing the common variations of a pattern implementation, the roles of the participating classes can be identified and the intent of the corresponding source code is then revealed.

The design patterns are getting more and more attention in forward engineering as well as in the reverse engineering community during the last decade, because they are based on important rational used by designers during the development of software. Patterns provide proven solutions and are helpful for the comprehension of large and complex legacy applications. They include expert knowledge, design decisions and contain intentions of designers, which are abstract in the source code. They become effective communication vehicle among designers. By recovering the instances of different patterns and then composing these small instances of patterns to large patterns, we can extract the core information of design model. Each design pattern has its unique intent, but it can play multiple roles when it is combined with the other patterns. Overlapping and composition of different classes and instances of the patterns with other patterns give information about the classes that play key role in different patterns. The recovery of overlapping and composition of design patterns is still an overlooked area for the design pattern research community.

Several methodologies, techniques and tools have been used for recovery of design pattern instances from legacy source code. Static, dynamic and semantic analysis approaches with manual, semi-automated and automated processes are used to recognize design patterns from different legacy applications. All approaches have their strengths and limitations. The static recovery approaches recognize pattern classes based on their (inheritance, association, composition, instantiation, generalizations, etc.), but these approaches fail to recognize some patterns whose structural signature is very weak or variable. The Bridge and Observer are cited as an example [7 8]. The static analysis tools are not able to handle all types of associations, aggregations, compositions and friend relationships in classes, which are important for pattern recovery [9]. Apart from these problems, the more serious constraints are the recovery of non-functional aspects (e.g. loose coupling between specific classes in a pattern). The dynamic analysis approaches cover areas such as performance optimization, software execution visualization, behavioral recovery and feature to code assignment. These approaches capture system behavior, but they are not practical in verifying the logic of a program. It complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. The semantic analysis approaches use the naming conventions, inline comments and design documents to supplement the static and dynamic analysis. Most of the pattern recovery approaches use different intermediate representations of source code (UML, AST, ASG, Parse trees, etc.) instead of directly using the source code. The choice of intermediate representation format directly affects the choice of the algorithms for

discovery [1]. We believe that the structural analysis should be used with low level analysis of code and annotations to narrow down the search space of detecting patterns.

1.2 Application of Design Patterns

Many patterns have been developed which are used in different areas such as software architectures, user interfaces, concurrency, security, services, etc. It is widely recognized that proper use of design patterns can improve software quality and development productivity [127]. Developers adopting patterns and practices can expect an average productivity increase of 25 to 40 percent, depending on their skill level and complexity of application [17]. Different open source, commercial and business applications have applied patterns very successfully and gained significant benefits in terms of improved application management, lower cost of maintenance, lower cost of updating and improved application performance, etc. Most GoF [13] patterns are applied in different open source and commercial applications and they build architectural frameworks. For example, Singleton and Factory method are used to implement java.awt.Toolkit in the Java AWT package (a GUI toolkit). Composite, Interpreter, and Visitor patterns form the basic architecture of Jikes (a Java compiler written in C++) [18]. The Composite pattern is also used in different applications such as (GUI containers and widgets, HTML/XML parsing, File managements, etc.). The Decorator pattern is used for developing different websites. The Template method is applied in different frameworks, e.g. Servlet's, EJBs, and Web Service Frameworks. Java 1.1 AWT event model uses Observer pattern. The Microsoft .NET Framework base class library used extensive application of patterns like Observer, Iterator, Decorator, Adapter, etc. Composite and Template methods are used in the structure of ASP.NET. The use of Command, Flyweight, Wrapper and Iterator improve the quality and the comprehensibility of XML applications. Similarly, (Flyweight, Strategy, Adapter, etc.) are used in Apache Ant (a Java build tool). J2EE application model is build by using different J2EE design patterns. All enterprise applications are divided into different tiers known as components. The intercepting filter, front controller, view helper, etc. are used to build the presentation tier of J2EE model. Service oriented architecture patterns are used for different business and web services. Microsoft patterns and practices provides .NET developers with the building blocks and guidance to rapidly build complex, loosely-coupled applications to meet their current business needs and a structured application environment that can be cost effectively adapted as needs change[11]. The combined application of Service Façade and UI Mediator yield the benefits of establishing behavioral regulators at front and back end. The increasing uses of web technologies foster web developers to use web application patterns. For example, Filter pattern is used to execute specific code conditionally at the start and end of page request. These applications reflect the attention of community towards automated

recovery of patterns from the legacy applications. The recovery of design patterns from existing legacy applications benefit and enrich other related disciplines like program comprehension, program maintenance, refactoring source code, code reviews, restructuring source code, source code validation, source code documentation and reengineering of applications.

1.3 Role of Design Pattern Detection Tools

Each design pattern recovery approach is assisted by one or more tools to validate the recovery process used for the recognition of different instances of patterns. Tools have been used for the pattern representation, recognition, combination, generation and application. Different approaches apply different tools to extract patterns from source code with different precision and recall rates. Most of the tools are language dependent because they are developed using hard coded algorithms, which make their application specific to single language and customization of tools become difficult. Some tools face problem of scalability and performance in case of very large and complex applications. Another category of tools depend on the results of third party tools, which effect their performance, flexibility, robustness and accuracy. The capability of a pattern detection tool is determined by its precision and recall rates, which depend on false positive and false negative rates. Another important factor that is considered to measure the accuracy and the completeness of tool is its ability to match different implementation variants of single design pattern. Singleton is a simple design pattern, but it has different implementation variants. Tools like FUJABA [97] and PINOT [98] are not able to detect all the variants of Singleton design pattern from the source code which is relatively simple to recognize. A good pattern matching tool should have low false positive and false negative rate. In general, recognizing program behavior is known as an undecidable problem, hence a fully automated static analysis will not be able to achieve 0% false positive and false negative rates [10]. Some tools provide visualization support, but we think that exact location of each discovered pattern is also important for maintenance activities. Due to limitations and problems in different pattern matching tools, we developed our custom build tool (UDDPR) to support and validate our approach used for the extraction of different implementation variants of design patterns from the source code of different applications.

1.4 Problems in Design Pattern Recovery

The recovery of design patterns from large and complex legacy applications started 14 years ago with the publication of Gamma et al.'s [13] prominent book, but some key questions still remain unanswered. A number of techniques have been used for recovery of design pattern instances from the legacy source code in the past. Each technique has its

strengths and limitations. The recovery of different instances of design patterns from source code becomes arduous due to the following problems:

I) Design patterns have different implementation variants and there is no standard formal definition for each design pattern, which is acceptable to the whole research community.

II) "There are several ways to implement the various relations in classes like delegation, aggregation, etc., which make pattern recovery process difficult".

III) "Some programming languages provide library classes which facilitate pattern implementation, but they complicate pattern recovery process".

IV) "Benchmark systems are not available which can compare and validate the results of different approaches".

V) The instances of different design patterns are scattered in the source code and there are no formal rules for their composition.

VI) Pattern descriptions are abstract, informal and are usually not documented in the source code.

VII) "Human opinion in pattern mining process is very desirable in some cases as it reduces the number of false positives".

VIII) Most approaches are language specific and they are not able to recover patterns from other languages.

IX) "Some approaches take the intermediate representation of the source code which affect the algorithms for pattern recovery".

X) Some approaches take only the single representation of design pattern and fail to recognize different implementation variants of the same pattern.

XI) Formal specification methods and languages focus only on the specification of single design pattern, while they do not address the composition and relationship of patterns with the other patterns. So the frameworks or architectures that use combination of patterns cannot be detected by existing approaches.

XII) Some approaches fail or their recognition precision is very low when the size of software to be examined for pattern recovery increases.

Some of the above mentioned problems are also highlighted by [7 6 5] as quoted above. These problems demonstrate many open issues in design pattern recovery and reflect the attention of reverse engineering and reengineering community towards automated detection of design patterns. We do not claim to address all of the above issues in this thesis, but we plan to overcome some of above mentioned obstacles with our pattern recognition approach. We focus on the variant pattern specifications, which help us to recover implementation variants of different patterns from the source code of multiple languages with the help of multiple techniques. There is still no agreed upon solution on design pattern recovery to-date, because each approach takes his own

definition for each design pattern. It causes inconsistency in the results of different approaches used in the literature [2 3 11 12]. So it is still premature to claim that design pattern recovery is fully automatic. It is a challenge for the reverse engineering and reengineering community to prove the automation of design pattern recovery.

1.5 Contributions

A number of existing approaches are used for design pattern recovery in the past and each approach used different search techniques. For example, the concept of database queries is used by [65 48 23 47] to recover patterns from legacy source code. These approaches first transform source code into a database structure using modeling tools or create database models directly from source code and then use queries to extract patterns. SQL queries are fast to extract pattern features, but are only limited to artifacts available in the database model. Modeling tools are not capable of extracting all the properties of source code in database models and are mostly restricted to static facts. So these approaches are not able to extract all patterns with SQL queries. Some approaches detected patterns using hard coded algorithms and they are not able to recognize the variants of different patterns. User cannot add new features in pattern definitions to control variation which is very common. Similarly, the numbers of approaches combine static and dynamic analysis to improve precision and recall, but they cannot detect the patterns which have similar static and dynamic features. These approaches fail to capture program behavior and intent, which is important to distinguish number of patterns. For example, State and Strategy have similar static and behavioral features but have different intents. This thesis combines multiple search techniques including SQL queries, Regular expressions and Source code parsers to improve the precision and recall for design pattern detection. With the help of multiple search techniques, we are able to extract pattern features from intermediate representation of source code as well as directly from source code using source code parsers. The concept of annotations supplemented searching methods to reduce the search space and time for design pattern detection.

The major contributions of this dissertation are as follows:

I) Integration of multiple techniques to detect the structural as well as implementation variants of design patterns from source code of multiple languages. Multiple searching techniques improve precision and recall of presented approach.

II) Customizable and reusable feature types to define patterns with different variations.

III) Creation of adaptable pattern definitions using feature types.

IV) A prototype tool developed an Add-In with Visual Studio.Net framework using Sparx Enterprise Architect Modeling Tool to extract patterns from source code of multiple languages such as Java, C/C++ and C#.

1.6 Structure of the Thesis

Chapter 2 discusses fundamentals of design patterns, types of design patterns, design pattern classifications, documentation styles for design patterns, design pattern specification methods and variants of design patterns. The chapter concludes that design pattern variants hinder the accuracy of pattern recovery approaches.

Chapter 3 elaborates the state of art for different pattern recovery techniques and tools. A recent review about the important attempts used in the past is presented and discussed. It discusses strengths and limitations of approaches used in the past. The wide disparity in results and limitations of current approaches become the motivation for our technique. The chapter concludes with the clear requirements which will be focus of this thesis.

Chapter 4 describes the main concept of proposed approach used for pattern recognition. Pattern recognition approach is divided into two phases. The goal of first phase is to create the pattern definitions. The concept of different searching techniques used for pattern recognition is discussed in second phase. The challenges and concepts used to handle them are discussed at end.

Chapter 5 describes the feature type definitions and pattern definitions. The major emphasis is on creating variant pattern definitions by using customizable and reusable feature types with static and dynamic parameters. The pattern definition creation process is illustrated with activity diagram. Examples of pattern definitions are illustrated at the end.

Chapter 6 explains pattern recognition process which is based on multiple searching techniques used for pattern recognition. The benefits and limitations of integrated searching techniques are highlighted. The objectives and the scope of approach are discussed.

Chapter 7 presents initial prototype UDDPD, which is used to detect different patterns from source code of multiple languages. The features and limitations of prototype are discussed for future extensions and improvements. Prototype features are compared with the tools presented in review in Chapter 3.

Chapter 8 evaluates approach using different case studies and compares it with state of the art approaches. It discussed the assumptions for evaluation of extracted results. It compares the scalability, precision and recall rates with the existing approaches. The approach discusses wide disparity in the results of different approaches. Furthermore, it discusses the threats and validity of extracted results.

Chapter 9 presents conclusions and outlines new ideas, which are outside the scope of this thesis for future work. The contributions are highlighted and possible future extensions are elaborated.

Appendix A gives the glossary of terms used in this thesis.

Appendix B describes design patterns using feature types.

Appendix C lists the features types based on SQL, Regular Expressions and Source code parsers.

Appendix D describes pattern definitions used in our prototype.

Appendix E lists the annotation used for the documentation and recovery of patterns.

Chapter 2

Fundamentals of Design Patterns

This chapter discusses fundamentals of design patterns and problem of variations in the implementation of design patterns, which poses challenges for new and existing design pattern recovery approaches. Section 2.1 discusses background, types, classifications and relationships between design patterns. The documentation styles of design patterns used in the past are presented in Section 2.2. Section 2.3 describes different specification methods used for description of design patterns. Section 2.4 discusses variants of different design patterns, which becomes the motivation for concept of customizable and reusable features types used for the definition of design patterns.

2.1 Background, Types and Classifications of Patterns

The current use of patterns has roots from work of architect Christopher Alexander who used patterns to improve process of designing buildings and urban areas in late 1970s. Patterns are now widely used for analysis, architecture, design, process, and documentation of applications. They are applied in art, crafts, buildings, architecture, manufacturing, services, leaderships and software, etc. The application of design patterns became popular in early 1990s due to ground breaking presentations in OOPSLA 94 [114] and with the publication of GoF book by Gamma et al. [13]. Currently, patterns are used in different areas such as software architectures, user interfaces, concurrency, security, web, services, etc. Alexander define pattern as “A recurring solution to a common problem in a given context and system of forces”. Gamma et al. [13] define design patterns as solutions to recurring problem in software system design that help in improving reusability, maintainability, comprehensibility, evolvability and robustness of legacy applications.

The use of patterns for development of applications started in 1990s, but the importance of patterns for reverse engineering was realized with publication of design pattern recovery approach presented by Krammer et al. [27]. It is misconception that design patterns provide concrete rules or idea that should be followed each and every time to solve a problem. A pattern gives description of expertise which leads to method how to solve a problem. It is very important to know when to use a pattern for particular problem. The wrong selection of pattern can cause adverse effects like slow and cumbersome development. It can make maintenance difficult and reuse inconsistent.

The following arguments give justification about the use of patterns for development and recovery of software applications:

I) "Designing object-oriented software is hard and designing reusable object-oriented software is even harder [13]".
II) Experienced designers reuse solutions that have been proved, tested and recommended by the academia as well as industry in the history.
III) Well-structured object-oriented systems have recurring patterns of classes and objects. The recovery of patterns yields benefits to program comprehension.
IV) The skill and experience of designers that have worked in the past give opportunity to maintainers and developers to be more productive during understanding of legacy applications.
V) Design patterns facilitate communication among designers, developers, and maintainers by providing a common vocabulary.
VI) Patterns make it easier to reuse successful designs and avoid alternatives that diminish reusability and hamper design understandability.
VII) The application of patterns improves the documentation of system as whole and design phase in particular.
VIII) Applications of design patterns facilitate design modifications and speed up development process by providing tested and proven development paradigms.
IX) Design patterns become basis for design standards and ensure consistency between the components.

2.1.1 Types of Patterns

The increased interest in using patterns for development of new applications fostered community to create new patterns, which should meet requirements of designers and developers. Patterns are divided into the following types.

Domain patterns: Domain patterns capture design solutions for family of applications in software domain. Systematic use of domain patterns can reduce development time of applications in a particular domain [140]. The examples of domain patterns are Remote Operation pattern, Cyclic Execution pattern, etc.

Architectural patterns: Architecture patterns are used at higher level of abstraction and are applied to a system as a whole. Different design patterns are used to compose an architectural pattern. For example, MVC uses Observer, Strategy and Composite design patterns.

Design patterns: A design pattern describes a typical design problem, and outlines an approach to its solution. Design patterns are used at middle level of abstraction to implement the subsystems or components of subsystems. Fundamental design patterns like (Delegation, Interface and Immutable, etc.) are used to implement design patterns.

Idioms: An idiom is low level pattern specific to programming language. Idiom describes how to implement particular aspects of components or relationship among them using feature of given language. Idioms are used to implement different design patterns.

Above types are based on the abstraction level used for application of patterns in different domains. The approach used in this thesis focuses on detecting design patterns.

2.1.2 Classifications of Patterns

In many disciplines, but most notably in the natural sciences, classifications are created, maintained, and improved to increase scientific knowledge [146]. The purpose of design pattern classification is to make convenient for the developer to select the right pattern for the particular problem. Different authors classify patterns based on multiple criteria. The GoF classification is still well know and adopted by most of the authors. We highlight the following well know classifications.

Martin Fowler [117] classification: The authors divide patterns into eight groups such as (Domain Logic Patterns, Data Source Architectural Patterns, Object Relational Patterns, Web Presentation Patterns, Distribution Patterns, Offline Concurrency Patterns, Session State Patterns and Base Patterns). These patterns are used for the development of enterprise applications.

Frank Buschmann et al. [116] classification: The authors classify patterns into architectural patterns, design patterns and idioms. The patterns are classified on the basis of purpose in each category. The consistency of using one category in another category is major problem in this classification.

GoF [13] classification: It is first well described and documented catalog of design patterns by Gamma et al. [13]. It consists of 23 design patterns which are classified into creational, structural and behavioral patterns. This classification is based on pattern purpose and scope. The purpose of pattern specifies what the pattern does. The scope specifies that a pattern applies to classes or objects.

Walter Tichy [115] classification: Walter Tichy catalogs over 100 general purpose patterns on the basis of categories: decoupling, variant management, state handling, control, virtual machine, convience patterns, compound patterns, concurrency and distribution. Each category is divided into subcategories. The author clarifies the ambiguities in GoF classification, which focuses only on object oriented design patterns.

Shi and Olsoon [25] classification: The authors re-classify GoF design patterns based on structural and behavioral resemblance into five categories: language-provided patterns, structure-driven patterns, behavior-driven patterns, domain-specific patterns and generic concepts.

Hammouda et al. [87] classification: The authors reclassify GoF patterns based on factors including purpose, scope, discipline, domain, paradigm and granularity.

Walter Zimmer [118] classification: The author analyzes the internal structure of GoF patterns with deeper understanding, but the classification tends to obscure the purpose of patterns.

Vandana Bajaj [119] classification: This classification is based on different levels and each level classifies pattern using abstraction on pattern features. The used classification improves pattern understanding using hierarchy of pattern features at different levels.

The above classifications of patterns help user to understand the purpose and application of each classification at various levels of abstractions. We focus on GoF classification of design patterns in pattern recognition process.

2.1.3 Design Pattern Relationships

Each pattern has specific purpose, but there are variability's and commonalties between different design patterns. Relationships between patterns give designer alternative for selection of a particular pattern to solve problems. For example, Proxy, Adapter, and Decorator have commonality that each provide interface to clients, but the type of interface vary for each pattern. Similarly, Singleton is used in Factory method, Abstract factory method and number of other patterns. The knowledge of relationships between design patterns can be used for defining formal composition relations between patterns and these (formal definitions) can be used for detecting compositions between design patterns during reverse engineering of legacy source code. The relationships between GoF patterns defined by Gamma et al. [13] are shown in Figure 2.1. A number of approaches have been presented by different authors, which focus on relationships between design patterns in different aspects.

Reference [21] has presented a pattern language which contains over 90 patterns based on large scale structure and small scale structure. Large scale structure consists of different pattern language fragments and each language fragment contains number of patterns and relationships between the patterns. The small scale structure defines relationships between patterns using constructs such as uses, conflicts and refine.

Walter Zimmer has presented a paper [118], which focused on defining relationships between design patterns based on different layers. Each layer defines relationship between specific categories of design patterns. Author claimed that presented relationships between patterns can be used for organizing existing patterns and new discovered patterns, which is not possible in Gamma classification given in Figure 2.1. The adopted classification layers can be used for understanding complex relationships between design patterns which is beyond the scope of [13].

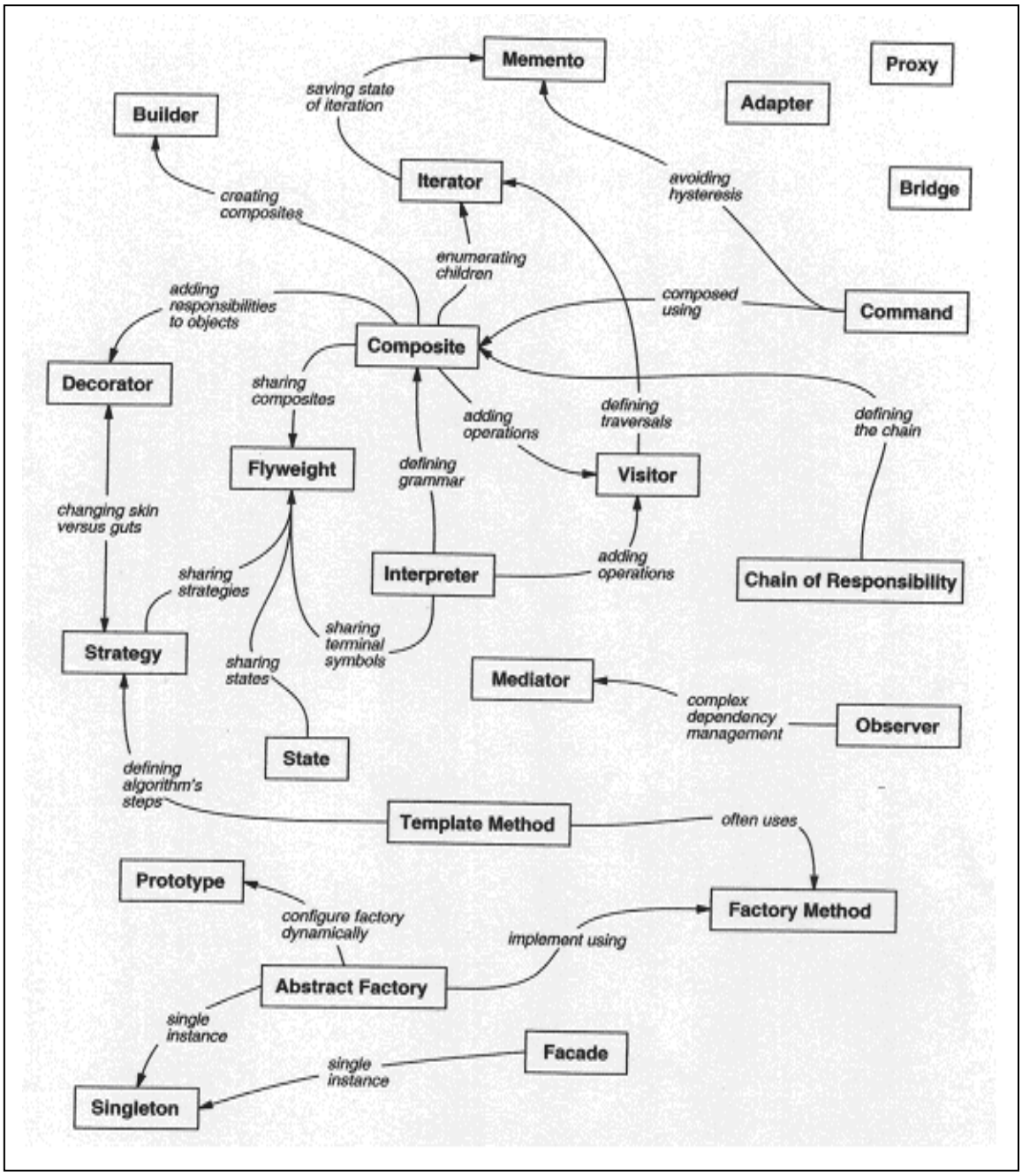


Figure 2.1: Relationships between GoF design patterns [13]

2.2 Documentation of Patterns

There is no single, well defined and standard format for documenting design patterns that is accepted by the entire design pattern research community. The variety of different formats has been used by different pattern authors and these formats have resemblance in essential common characteristics. However, according to Martin Fowler certain pattern

forms have become more well-known than others, and consequently become common starting points for new pattern writing efforts [85]. All the pattern documentation forms discuss only basic characteristics of each pattern while possible variations are rarely discussed in detail, which are important for the specification, documentation and recovery of different variants of patterns. Table 2.1 lists different formats used by different authors for the description and documentation of design patterns in the past.

Table 2.1: Documentation forms of design patterns

Pattern form	**Characteristics**
Alexandrian Form[88]	Title, Problem, Discussion, Solution, a Diagram, Prologues and Epilogues.
Canonical Form[88]	It is more formal and complete than the Alexandrian form, containing additional sections on Context, Forces, Resulting Context, Rationale and Known Uses.
GoF Form[13]	Name, Alias, Problem, Context, Forces, Solution, Example, Resulting context, Rational, Known uses, and Related patterns.
Compact Form[88]	Context, Problem, Forces, Solution and Resulting Context.
Cockburn PM Form[88]	Title, Thumbnail, Indications, Contraindications, Forces, Do this, Side Effects, Overdose Effect, Related Patterns, Principles, Examples and Reading.
James Maioriello Form[88]	Name, Intent, Also known as, Motivation, Applicability, Structure, Participants, Collaboration and Consequences.
Beck et al.[94] Form	Title, Context, Problem, Forces, Solution and Resulting Context.
Fowler Form[13]	Title, Summary and "the bad stuff".
Kamyar Form[88]	Title, Problem, diagram, participants and Example.
Coplien and Schmidt[117] Form	Name, Problem, Context, Forces, Solution, Resulting Context and Rationale.
Holzner Form[93]	Intent, Problem, Discussion, Structure, Example, Checklist and Thumbnail.
Thomas Form[89]	Requirements, Icon, Summary, Problem, Solution, Application, Impact, Relationships, and Case study Example.
Buschmann Form[95]	Summary, Example, Context, Problem, Solution, Structure, Dynamics, Implementation, Example resolved, Variants, Known uses, Consequences, and See also.
Grand Form[92]	Synopsis, Context, Forces, Solution, Implementation, Consequences, API usage, Code Example and Related Patterns.
Bishop Form[91]	Role, Illustration, Design, Implementation, Example, Use and Exercises.
Vora Form[71]	Name, Problem, Solution, Why, How and Related design patterns.

2.3 Design Pattern Specifications

Design patterns are typically described by documentation in terms of several aspects, such as intent, motivation, structure, behavior, sample code, and related patterns. The specifications and descriptions of design patterns are very important for their successful implementation and recovery. Experienced designers and programmers reuse existing

tested solutions that have saved their time and efforts in the past. Traditionally, patterns are specified by using informal English and class diagrams. Most of the pattern writers use a combination of textual descriptions, graphical notations [70] and sample code fragments to describe each pattern. The structural pattern specification is the core for the specification of patterns, because it specifies the structural properties which are further used for describing dynamic features of patterns. Specifying pattern solutions at the UML metamodel level allows tool developers to build support for creating patterns and for checking conformance to pattern specifications [103]. Figure 2.2 describes the role of design pattern specifications in forward and reverse engineering.

Formal, informal and semi-formal methods are used for the specification of design patterns. Each specification method has its own advantages and disadvantages. The overview about these specification methods is given in the following subsections.

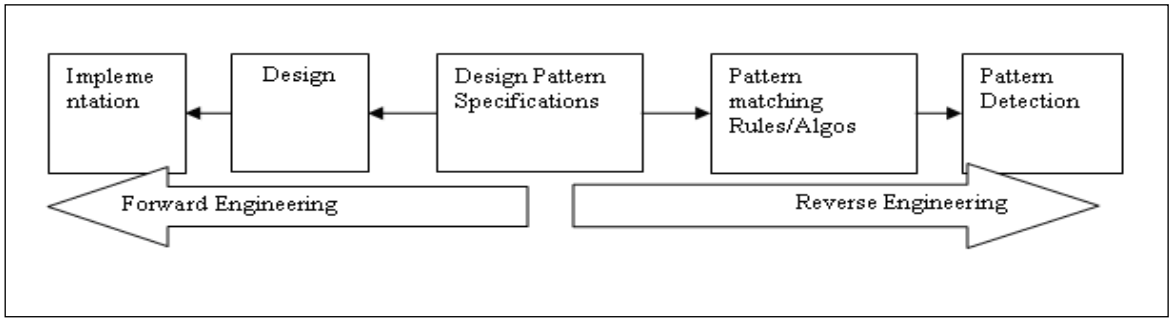


Figure 2.2: Pattern specifications in forward and reverse engineering

2.3.1 Formal Specification of Design Patterns

Design patterns are formally specified due to better understanding of individual pattern, automatic tool support for code generation, composition with the other patterns, evolution of patterns, reasoning, formal verification of solution and pattern recovery. Formal specification languages such as Z, VDM, RAISE, B and PVS are used, which are supported by the automated tools to verify the specifications. Eden [102] devised from scratch a new graphical language LePUS for the purpose of modeling design patterns.

A large and complex legacy software system design may consist of many design patterns. There are many reasons to check the consistency of the composition of design patterns. For example, when two patterns are combined, they may share some common parts. The part that they are sharing can play one role in one pattern, but another role in the other pattern. This situation may lead to an inconsistent combination. Formalization not only forces to be rigorous in the specification of the design patterns, but also offers a way to understand the differences between alternative specifications [71]. However, formal specification techniques focus only on the fundamental specification of a pattern instead of concentrating on the variable characteristics of design patterns, which are important for implementation and recovery. Secondly, it is difficult to convert formal

specification of patterns into our feature types, which are easy to understand by novice user for the purpose of customization. The feature types are described in Chapter 5. Thirdly, it becomes difficult/impossible to make changes in specification during design and implementation phases. Although, the formal specifications have some benefits in terms of defining fundamental definitions and are especially helpful for the composition of design patterns. Table 2.2 summarizes the most important techniques used for the formal specification of design patterns.

Table 2.2: Major formal methods for design pattern specifications

Specification Method/Language	Tool	Specification Aspect(**SPS[1]**)	Purpose	Specification Scope **(SPS[2])**
DPML	DP Tool	SPS+DPS	MDA(Model Driven Architectures)	SPS
RSL	RAISE Tools	SPS+DPS	MDA, Code verification	SPS
DisCo	DisCo tool	DPS	Behavior collection, code verification	SPS
RTPA	JACK Framework+TAPA	SPS+DPS	Expressive means for pattern description	SPS
Ocsid	Disco toolset +PVS theorem prover	SPS+DPS	Structuring Superposition Compositions	SPS+CPS
SPINE	Hedgehog	SPS	Code verification	SPS
RML	CrocoPat 2.1[72]	SPS	Analyze graph models	-
BPSL	BPSL Parser	SPS+DPS	Inference in general	SPS
Prolog		SPS+DPS	Pattern Repository	SPS
FOL	Proof Builder	SPS+DPS	Modeling, theorem proving	SPS
RBML	RBML-PI	SPS+DPS+SMPS	MDA	SPS
LePUS3	TTP Toolkit	SPS	Pattern Repository	SPS
TLA	DePMoVe	SPS+DPS	Writing predicates, state functions, actions and reasoning	SPS+CPS
PEC	PEC tool	SPS	Enforcing compiler	SPS
LOTOS	LOTOS simulator	DPS	Pattern composition Behavioral consistency	SPS+CPS
VDM	VDM-SL	SPS+DPS	Verification and generation of code	SPS
Logic theorems(FOL +TLA+Prolog)	Prolog Queries	SPS+DPS	Verification, composition and evolution of patterns	SPS+CPS
PVS	PVS Theorem prover	SPS	Proof checking, modeling checking	SPS

SPS[1]: Static Pattern Specification **DPS**: Dynamic Pattern Specification **SMPS**: State Machine Pattern Specification **SPS[2]**: Single Pattern Specification **CPS**: Composite Pattern Specification

2.3.2 Informal Specification of Design Patterns

The representative GoF book is the best example for the informal description of design patterns because of clarity and global understanding of design patterns. The flexibility of

using textual descriptions and informal diagrams attract developers for the easy understanding, but such specification is often ambiguous, imprecise, unavoidable to contradictions and not the best method to handle patterns at design times. These specifications are very lengthy, descriptive and lack support of abstraction, uniqueness and validation. Informal descriptions obstruct in the composition of design patterns, and it is also difficult to give reasoning using informal description, which is important for describing properties of some patterns. These specifications hamper tool support and there is minimal or no automated tool support for processing informal descriptions. When informal description of a design pattern is translated into implementation by different methods, it causes some errors in the implementation. Furthermore, Tabi et al. [73] highlighted that it is difficult or sometimes impossible to answer the following questions related with design patterns using informal specifications: Is one pattern the same as another (duplication)? Is one pattern obtained from a minor revision of another (refinement)? Are two patterns unrelated (disjointness)? These features are also critical as new patterns are being discovered, discussed and debated about.

The informal specifications of design patterns are mentioned in [74 70 75 76]. Moreover, Section 2.2 mentioned various forms used for informal descriptions of design patterns.

2.3.3 Semiformal Specification of Design Patterns

Semi-formal methods focus in creating models of systems at each stage and have ability of automatic model transformation. They use graphical notations which abstract the detail of different activities. These techniques focus toward reducing the complexity and describe the specifications keeping in view the factor of easy human oriented understanding of the patterns. Most of the semiformal specifications are based on UML [76 77 78 79 80 81 82 83 84 7 103]. UML is a family of modeling notations for specifying, visualizing, constructing, and documenting artifacts of software-intensive systems [41]. The specifications based on UML are supported by different tools, which can extract the static and the dynamic features of specifications, but they do not provide information on high level concepts such as intent, usability and consequences. The instructions inside certain activities may be in natural language. These approaches use top down structure and XXM [90] models for writing specification. XXM models make possible for machine states to match input/out screens and different transition can match processing of individual inputs. They are scalable and let the user free to have the knowledge of discrete mathematics.

2.4 Challenging Problem for Design Pattern Recovery

The structural as well as implementation variants of design patterns hamper accuracy of pattern recovery approaches. While one can define the pattern, but still there are no standard catalogs for the variants of each pattern. The structural variants of design patterns are relatively easy to detect if the new structure of same pattern is already documented. The variations of design pattern definitions are discussed in Chapter 5. The approaches based on hard coded algorithms do not let user to customize pattern definitions, which can match with different variants of same pattern. The applications of modern programming features facilitate the implementation of patterns, but they pose challenges for existing and new pattern recovery approaches. The implementation variants of design patterns are on the mercy of developers. We used the concept of customizable pattern definitions which allow user to refine pattern definitions. The question rises whether user is able to understand and refine existing pattern definitions? It depends on the specification method and user knowledge. Mostly, users are not able to understand specifications, which are based on pure formal techniques. We used feature types, which are comprehensible by the user to customize pattern definitions. The feature types and pattern definitions are discussed in Chapter 5. We highlight the problem of variants using Singleton, Factory method and Proxy patterns.

2.4.1 Singleton Variants

Singleton is considered simplest pattern but different implementation styles of Singleton make its recovery difficult. Most pattern recovery techniques based on structural and dynamic analysis are not able to detect all the possible variants of Singleton. Singleton is mostly used in Builder, Prototype, Factory method and Abstract factory. The uses of Singleton in number of patterns make its recovery more important. Tools based on hard coded algorithms are not able to detect all the variants of Singleton. Different implementation variants of Singleton are discussed in [96]. We discuss the variants of Singleton by the following examples.

```
public class  product {
private static product  instance = new
product();
product() {}
public static product getInstance() {
return instance;}
}
public class  Apply {
public useS1 () {product s1 = new
product() ;}
}
// class Product is not singleton
```

```
public class S3 {
private static S3 instance;
private S3 () {}
public static S3 getInstance1() {
if (instance == null) instance = new
S3();
return instance;
}
public static S3 getInstance2 () {return
new S3 ();}
}
// class S3 is not singleton
```

Figure 2.3: Sample codes similar to singleton in java

Figure 2.3 describes source code examples which partially match the properties of Singleton, but the state of art tools such as Fujaba [97] and PINOT [98] report false positive and false negatives for these instances. Reference [96] explains that why these are not true Singleton instances. Similarly, most of tools are not able to detect Singleton instances mentioned in Figure 2.4 as variant of Singleton only with the exception of approach [15], but the tool used for realization of approach is not available publicly to validate its conformance. The instance in left side of Figure 2.4 shows the subclass implementation of Singleton using hashtable for creating the static instance for the Singleton. In such case, the static instance is managed by some other class. Most of the approaches do not take into consideration this variant and return false negative. The right hand side of Figure 2.4 describes the implementation of a Singleton when singleton instance is managed and accessed through different class. In such case, another class combines a placeholder with access point function. The Singleton getInstance method returns the instance of Singleton not through singleton class but with placeholder of different class.

Finally, Figure 2.5 explains subclass implementation of Singleton that is useful when we want to change the behavior of the singleton class. This implementation is used when singleton class has two subclasses: singleton 1, singleton 2 and we want to instantiate only one subclass of singleton either singleton1 or singleton 2. The tool DPDv4.3 [125] is not able to detect this variation. Such variation can only be detected when it is implemented within the definition of pattern or existing definition can be customized in pattern detection tool.

```
public abstract class Singleton {
private static Hashtable instances =
new Hashtable();// should this not be
//protected? –
public static Singleton getInstance()
throws Exception {
 if (!instances.has_key(getClass())
            instances.put(getClass(),
getClass().createInstance());
          return
instances.get(getClass());
        }
```

```
public class Singleton {
// Private constructor prevents instantiation
from other classes
 private Singleton() {}
 /**
 * SingletonHolder is loaded on the first
execution of Singleton.getInstance() */
 private static class SingletonHolder {
 private static final Singleton INSTANCE =
new Singleton();
  }
  public static Singleton getInstance() {
   return SingletonHolder.INSTANCE;
  }}
```

Figure 2.4: Subclass singleton with hashtable and different placeholder

```
public abstract class MazeFactory {
// The protected reference to the one and only instance.
protected static MazeFactory uniqueInstance = null;
// The MazeFactory constructor.
// If you have a default constructor, it can not be private
// here!
protected MazeFactory() {}
// Return a reference to the single instance.
public static MazeFactory instance() {return uniqueInstance;}
}
public class EnchantedMazeFactory extends MazeFactory {
// Return a reference to the single instance.
public static MazeFactory instance() {
if(uniqueInstance == null)
uniqueInstance = new EnchantedMazeFactory();
return uniqueInstance;
}
```

Figure 2.5: Variant of singleton

2.4.2 Variants of Factory Method

Factory method pattern is used for creating class of products without specifying the class creating those products. The essence of the Factory method pattern is to "Define an interface for creating an object, but let the subclasses decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses [13]". It is used in different toolkits and frameworks. Several approaches return many false positives during recovery of Factory method, because it has many implementation variants. More than nine structural variants of Factory method are reported by [24].

We discuss three possible variants of Factory method shown in Figure 2.6. The left side of Figure 2.6 shows standard GoF structure of Factory method, which is commonly known and recognized. The second variant of Factory method is shown in middle of Figure 2.6 which differs from first that Creator class creates the instance of concrete product class and concrete creator only return the concrete product. The approaches based on hard coded algorithms are not able to detect such simple variation and report number of false negatives. The third variant shown in right side of Figure 2.6 has additional composition between creator and product classes, which requires additional feature to specify composition between creator and product. This structure is similar to standard GoF structure with the addition of composition between creator and product. Similarly, the parameterized Factory method which allows Factory method to create multiple copies of objects and other variants can be specified by adding/removing features in the standard specification of Factory method.

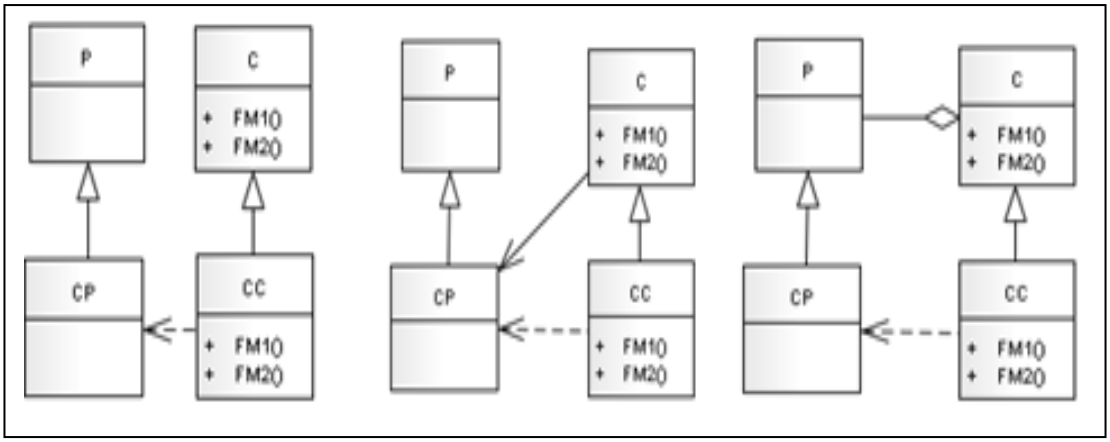


Figure 2.6: Variants of factory method

2.4.3 Variants of Proxy

Proxy can be used to provide interface to any other resource when it becomes expensive to create the duplicate copy of that resource. It supports the client server communication in the hybrid environment [99]. The request is forwarded from proxy to real subject according to kind of proxy such as remote, virtual, protective, cache, count and smart proxy. The Proxy and Decorator have the same structure but they have different purpose.

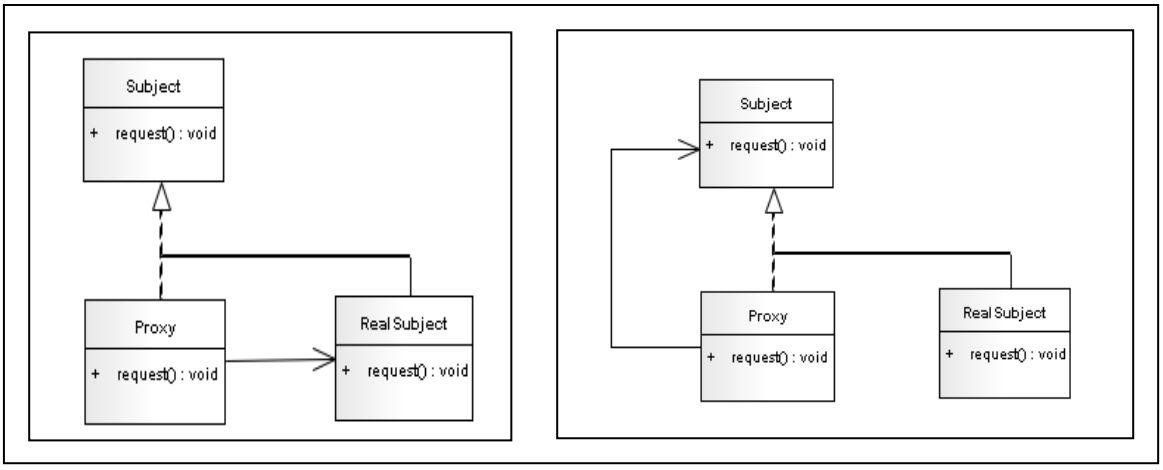


Figure 2.7: Proxy variations

Figure 2.7 shows the standard GoF structure of Proxy on the left. The right hand side of Figure 2.7 illustrates the variant of Proxy in which association is between proxy and subject classes. The delegation call is from proxy to subject class. The last variant of proxy is shown in Figure 2.8 in which proxy class has aggregation relation with real Subject class. Such variations create challenges for pattern recognition tools to handle different variants of same pattern.

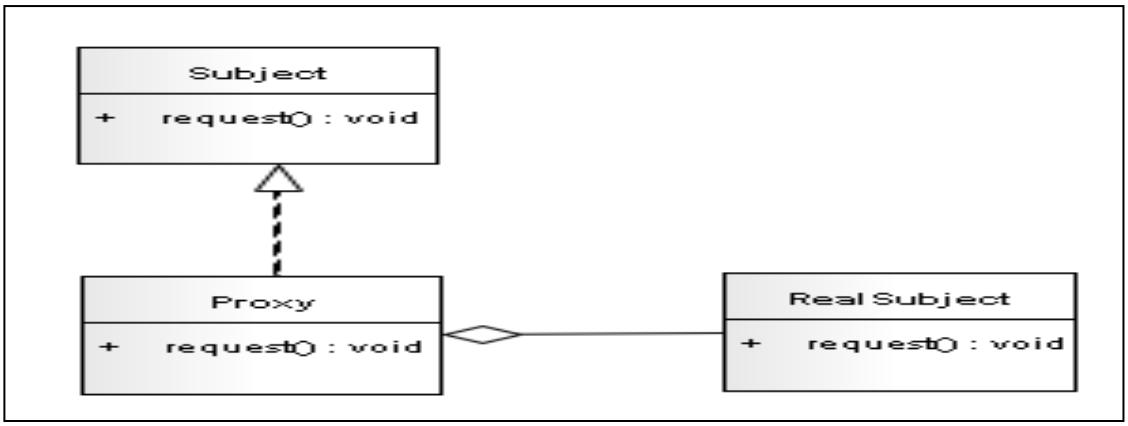


Figure 2.8: Proxy variation with aggregation

Detection of different implementation variants of same design pattern is major concern for design pattern recovery approaches. In order to handle variants of patterns, we used the concept of reusable and customizable feature types. The feature types use different recognition technologies to detect the variants of different design patterns which are discussed in Chapter 5.

2.5 Summary

This chapter discussed the basic concepts of design patterns, including pattern types, pattern classifications, pattern documentation, pattern specification and problem of variants for design pattern recovery. Section 2.1 discussed the fundamentals of design patterns. The styles used for documentation of patterns are discussed in Section 2.2. Section 2.3 discussed the specification methods used for the description of patterns with strengths and limitations of each method. The major focus of chapter was to highlight the problems of variants for different design patterns and Section 5.3 discusses how our approach describes variants. The variants of Singleton, Factory method and Proxy are discussed in Section 2.4.

Chapter 3

Related Work

This chapter provides an overview of the state of art in the area of design pattern recovery and discusses strengths and limitations of existing approaches. Section 3.1 discusses the review of current approaches used for design pattern recovery. Section 3.2 highlights the major tools and their features used in the pattern recovery approaches. Section 3.3 discusses about the disparity of results discovered by different approaches. Section 3.4 discusses critical review and observations on the results of different recovery approaches and recommends guidelines based on observations. Finally, Section 3.5 lists the requirements based on state of art work presented in this chapter and problems discussed in Sections 1.4 and 2.4.

3.1 Review of Pattern Recovery Approaches

In the nutshell, pattern recovery approaches are classified into structural analysis, behavioral analysis, semantic analysis and formal specification/composition analysis to recover patterns from the source code of different legacy applications. Structural analysis approaches are based on recovering the structural relationships from different artifacts available in the source code. They focus on recovery of structural design patterns such as Adapter, Proxy, Decorator, etc. Behavioral analysis approaches take into account the execution behavior of the program. These approaches use dynamic analysis, machine learning and static program analysis techniques to extract behavioral aspects of patterns. Semantic analysis approaches supplement structural and behavioral analysis approaches to reduce the false positive rate for recognition of different patterns. The semantic analysis approaches use naming conventions and annotations which contain key information about the classes and methods in the source code. Semantic analysis becomes important for recovery of patterns which have similar static and behavior properties. For example, Bridge and Strategy patterns have the similar structural and behavioral characteristics. The semantic analysis can be used to differentiate intent of these patterns. The formalization of design patterns is another important area that we take into consideration during our review because some approaches extract patterns from source code based on formal specification of design patterns. Formal specification of patterns is also important for composition of different patterns. A good review of different techniques used in the past is presented in [1 19]. The authors have presented the deep insight view of work starting from infancy

of design pattern recovery to the publication of this article. Due to everyday breakthrough in the field of information technology and presentation of new methodologies used for recognition of patterns, we focused our attention on the recent methods presented in literature. The review presented in [1] also misses some key information about different approaches like precision and recall rates, which are important to measure the accuracy and completeness of pattern recovery approaches. The approach only compares his self results with [20]. We focus our review on existing approaches starting from 2005 to onward because; we think that new approaches have improved their concept of pattern recovery by incorporating new emerging advances in technology and methods. We take some facts from [1] in our review and new approaches are given more attention, which have discovered patterns from different legacy applications very successfully. Tables 3.1 and 3.2 summarize/compare the features and results of different structural, dynamic and semantic analysis approaches. A review about major contributions and strengths/limitations of different approaches is discussed in the following subsections.

3.1.1 Structural Analysis Approaches

These approaches focus on inter-class relationships to identify the structural properties of patterns, but they completely miss the behavioral aspects. Structural analysis approaches explore relationships: class inheritance, associations, friend relationships, interface hierarchies, modifiers of classes and methods, method parameters, method return types, attributes, data types, etc.

Some of structural analysis approaches extract inter-class relationships from the source code using different third party reverse engineering tools and then perform pattern recognition based on extracted information. For example, reference [7] parses the source code using third party commercial tool called Understand for C++ [22]. Tool extracts the entities and the references from C++ source code and stores information in a database. In the continuation, queries are performed on the database to extract different properties of patterns. They recovered Singleton, Factory method, Template method, Observer and Decorator from VCS (Version Control System). The experiments are performed on VCS containing only125 classes that are not available publicly and scalability of approach is questionable.

The reference [23] extracted inter-class relationships from Java source code and store information in a program metamodel. Queries are executed on the extracted information to match different creational patterns. The authors used D^3 (D-cubed) tool for detection of Singleton, Factory Method, Abstract factory method, Builder and Prototype from Java source code. The approach establishes the program metamodel to formulate the definitions of design patterns in first order logic. They translate the logic formulae to SQL and execute the queries against a database containing program metamodel, which is manual, laborious and error prone activity. The approach has detected some non-standard variants

Table 3.1: Comparison of different pattern recovery approaches

Authors	Tools	Techniques	Lang-uages	Anal-ysis	System Representation	Case studies	P%	R%
Gueheneuc et al. [66]	PTIDEJ	Explanation based with Numerical Signatures	Java	ST&SE	PADL, XML	JUNIT v3.7 JUZZLE v0.5 RISK v1. 0.7.5 JSETTLERS v1.0.5 JTANS v1.0 etc.	70 21	100 59
Pattersson et al. [67]	CrocoPat	Database Query	Java	ST	RSF	SWT 3.1.0 Swing 1.4.2	NM	75
Lucia et al. [19]	DPRE	XPG formalism &LR based parsing	Java	ST	SVG, AOL, AST	JHotdraw5.1, JHotdraw6.0b1, QuickUML2001, Apache Ant etc	62 - 97	NM
Alnusair and Zhao [50]	PUT(under development)	OWL	Java	ST&SE	SWRL	JHotDraw 6.0b1 JUnit3.7 AWT	61 100 65	NM
Gueheneuc et al. [45]	DeMIMA	Constraint Solver	Java	ST&BE	AST& UML models	JHotDraw 5.1, JUnit3.7, Quick UML2001	43.2 43.8 34.8	100
Sartipi and Hu [42]	Eclipse plug-in	Mixed (FA,SPM,LA)	Java	ST&BE	PDL	JHotDraw	NM	NM
Kirasic and Bash [33]	NM	OWL	C	ST	AST	C examples	NM	NM
Stencel and Wegrzynowicz [23]	D^3	Database Query	Java	ST&BE	FOL	JHotDraw 6.0b	NM	NM
Arcelli et al. [47]	JAPADET	Database Query	Java	ST&BE	XML	JAPADET and other examples	65.6	NM
Dong et al. [35]	DP-Miner	Matrix and Weight	Java	ST&BE&SE	XMI	JAWT JEdit, JHotDraw etc.	NM NM 95.7	89.2
Kaczor et al. [64]	Ptidej	Bit Vector algorithms	Java	ST	Bit Representation	JHotDraw 5.1 QuickUML 2001	50 15	NM
Shi and Olsoon [25]	PINOT	Data/Control Flows	Java	ST&BE	AST	Apache Ant, JHotDraw etc.	NM	NM
Tsantalis et al. [20]	DPD v4.3	Similarity Matrix	Java	ST	Matrix	JHotDraw 5.1 JUnit3.7, JRefactory 2.6.26	95.6 100 91.7	100
Wang and Tzerpos [65]	DPVK	REQL query	Eiffel	ST&BE	REQL static & RSF dynamic	Eiffel system Student projects	NM	NM
Philippow et al. [3]	Plug-IN	Minimal Key Structure	C++	ST	Class Hierarchy	Student projects	100	100
Costagliola et al. [29]	DPRE	XPG formalism &LR based parsing	C++	ST	SVG, AOL, AST	Galb++ Lib++, Socket, Mec	83.3 100 100 100	100 NM 100 100
Ferenc et al. [36]	Columbus	Machine Learning	C++	ST	ASG, XML DOM tree	StarWriter	67-95	NM
Smith and Stotts [34]	SPQR	Rho-calculus	C++	ST	OML, OTTER,XML	TrackerLib KillerWidget, Namespace etc.	NM	NM
Arceli and Cristina [46]	MARPLE	Data Mining	Java	ST&BE	XML	ADTree BayesNet, Prism etc.	76.4	63

NM: Not Mentioned FL:First Order Logic PDL: Pattern Description Language DPD: Design Pattern Detection

of design patterns, which are not detected by the tools FUJABA [97] and PINOT [98]. The metamodel used by approach supports only Java language and creational design patterns. The applied tool is not available for validating the results of approach.

Pettersson et al. [67] have presented an approach to detect Singleton and Observer from (SWT 3.1.0 and Swing 1.4.2) using static analysis. The approach specifies patterns using first order logic and then apply Crocopat tool [145] for querying the relational facts from program structure and static semantics. The major focus of approach is to set gold standards for evaluating the accuracy of different pattern recovery approaches using fine-grained metrics. Authors proposed the impedance of benchmark systems for evaluating accuracy of different pattern recovery approaches. It is difficult to generalize approach for all patterns, which cannot be extracted accurately without dynamic analysis.

The approach presented in [3] used automatic pattern recovery technique using minimum key structure. Authors get intermediate representation of the source code using Rational Rose [144]. The capability of the approach depends on the extraction capability of the Rose tool. The approach is not customizable, because it used algorithms that are hard coded in the source code. Authors claim improved precision and recall values on locally developed software applications which cannot be validated. The false positives are returned in the case of different implementation variants of patterns. Although, authors claim the extraction of all GoF [13] patterns with precision of 100% from Java source code examples developed locally.

Reference [33] has presented an ontology based architecture for design pattern recognition. The approach uses parser, OWL ontology's and an analyzer. The approach integrate knowledge representation field and static code analysis for pattern recognition, while it lacks support for dynamic analysis, which is important for discovery of behavioral patterns. Secondly, the authors discussed only the discovery of Singleton pattern which is simplest pattern for recovery.

Costagliola et al. [29] have presented a visual language based pattern recovery approach with different case studies. The experiments are performed on C++ examples (Galib++, Libg++, Mec and Socket). The approach extracted Adapter, Bridge, Composite, Decorator and Proxy with good precision and recall. The approach has performed experiments on small examples with few patterns and cannot be generalized. The two phased approach in [30] by the same group recovered design pattern instances at a course grained level by considering the design structure using visual language parsing technique. The identified patterns are then validated by a fine-grained source code analysis. The experiments are performed on JHotdraw (5.1 and 6.0b1), QuickUML 2001 and Apache Ant 1.6.2 to recover Adapter, Bridge, Composite, Façade, Proxy and Decorator. The authors have extended their approach in [19] on the same case studies. The extended approach first constructs a UML class diagram which is represented in SVG format. They map class diagram with the visual language grammar to detect different patterns. They

compared their results with the other approaches, but we have noticed that extracted pattern results are same in both approaches. They explained the implementation variants and possible composition of some patterns with other patterns, but these concepts are not implemented in their approach. The limitation of approach in its current implementation is also to handle the multiple inheritances. Their extracted pattern instances have also too much deviation from [20 25]. The deviation problem is discussed in evaluation Section 8.5.

Washizaki et al. [31] presented a structured analysis based technique using forward and backward method. The forward method checks whether the design pattern might has been applied and the backward method is used when the target design pattern to be detected is given. The approach focuses on detecting different design patterns with similar structure. The used technique helps the maintenance programmers in which version of program the target pattern was used. The experiments are performed on Java source code examples on State and Strategy patterns. The approach cannot be generalized for all patterns and it does not give any information about applied tool to validate results of approach.

Silva et al. [32] have used combination of formal methods and functional strategies to reverse engineer the Graphical User Interfaces of legacy applications. They focus on formalizing the Graphical User Interfaces for the interactive applications. The approach first extracts the interactor based models and the event flow graphs which are used to extract only the abstract model of user interfaces from legacy source code. The approach is not generic, recovers only limited number of patterns and is limited only to Java programming language.

The approach presented in [20] used static analysis technique for automatically detecting (Adapter/Command, Composite, Decorator, Factory method, Observer, Prototype, Singleton, State/Strategy, Template method, and Visitor) patterns from source code of different applications. Authors represent examined software as graph and use matrix to represent relationships between the source code artifacts. The proposed methodology used similarity algorithms to cluster of hierarchies, which reduce the search space for pattern detection. The approach extracts patterns quickly from medium size of software systems, but it suffers problem of scalability when the size of matrix increases. Authors claim 100% precision and recall on the examined examples. The analyzed disparities in the results of approach detected by number of approaches [1 19 45 111 147] make the claimed results suspicious.

Smith et al. [34] have presented a formalized design pattern detection and architecture analysis approach using SPQR toolkit. The authors recover patterns from C++ source code. The approach focused on elemental design patterns and explains recovery process with only Bridge pattern. The examples used for experiments are not available for validation.

Table 3.2: Patterns detected by different Approaches

Reference	Singleton	Abstract F	Factory M	Builder	Prototype	Adapter	Proxy	Bridge	Decorator	Composite	Facade	Flyweight	COR	Command	Iterator	Interprete	Visitor	Mediator	Observer	Memento	State	Strategy	TM
[52]		x				x				x													
[67]	x																		x				
[19]						x	x	x	x	x	x												
[46]	x									x							x		x				x
[45]	x	x	x		x	x			x	x				x			x		x	x	x		x
[42]						x	x		x										x		x	x	
[33]	x																						
[5]	x	x	x	x	x																		
[47]													x				x		x				
[35]						x		x		x											x	x	
[25]	x	x	x	x		x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x
[20]	x		x		x	x		x	x	x									x		x	x	x
[64]		x								x													
[20]						x	x	x	x	x				x									
[36]						x															x	x	
[3]	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
[68]										x											x	x	
[65]	x				x						x		x										x
[34]								x															
[40]	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x
[46]													x	x	x	x	x	x	x	x	x	x	x
[39]	x	x	x	x	x	x			x	x				x				x	x		x	x	x
[22]	x		x						x										x				x
[31]																					x	x	

3.1.2 Behavioral Analysis Approaches

The structural analysis approaches are unable to identify patterns accurately that are structurally identical or have weak structure. These approaches are supplemented by the dynamic analysis approaches to recover different patterns. For example, State and Strategy patterns are structurally identical. Similarly, Chain of responsibility, Decorator and Proxy have similar structures. The behavioral analysis deals with small number of classes and gives many false positives when the number of execution traces significantly increase. The major difficulty in behavioral analysis is that there can be various possible implementations for the same expected behavior [35]. The behavioral analysis techniques have used machine learning, dynamic analysis and static program analysis techniques to extract patterns from number of legacy applications.

Machine Learning

These approaches used decision trees, neural networks, fuzzy logic, etc. for pattern recognition. They focus on improving the accuracy of pattern recognition by training pattern recognition tools to identify the correct implementation variants of different design patterns. Most of these approaches are semi-automatic and require human intervention at some stages for pattern recognition. The overview of some important techniques is given below:

Frenc et al. [36] applied machine learning approach for design pattern detection that uses decision trees and neural networks for the detection of Adapter and Strategy patterns from the Starwirter [136] source code. The approach incorporates machine learning techniques to train its pattern recognition tool. They filter out number of false positives by using machine learning algorithms. They combined their previous static pattern mining technique [28] with machine learning technique to reduce number of false positives. Authors claim precision value of 67-95% on different examples. The approach provides good theoretical background, but it is applied only on Adapter and Strategy patterns. It is difficult to realize the generalization of approach for all GoF patterns, because the tool used for experiments is not available publicly.

Ding et al. [38] presented a compound record clustering algorithms approach for design pattern detection using decision tree learning. They applied decision tree algorithms to the compound records learning problem for design pattern detection by learning the classification rule for composite training examples from a set of collected records efficiently. Their method reduces the size of training examples and efficiently generates training examples from the software design. The approach focuses only on extracting sequentially connected relationships and detection of Strategy pattern is illustrated.

The reference [39] presented a finger print design pattern detection technique which follows the steps: repository creation; matrix extraction; rule learning; rule validation and interpretation. They proposed an experimental study of classes playing roles in design motifs using metrics and machine learning algorithms to fingerprint design motifs roles. Another focus of their study was to assess whether programs implemented using design motifs conform with software engineering principles generally. They performed experiments on (JHotDraw 5.1, JRefactory 2.6.24 and QuickUML 2001, etc.) to detect Abstract factory, Adapter, Builder, Command, Composite, Decorator, Factory method, Observer, Prototype, Singleton, State, Strategy, Template method, and Visitor patterns. The approach focuses on reducing search space, but it may return large number of false positives when certain roles are removed for detecting patterns.

Arceli et al. [46] have presented a design pattern detection approach which is based on supervised classification and data mining techniques to extract behavioral design patterns. They used multi-label approach to create networks and performed their experiments using

feed-forward neural networks and with back-propagation. The MARPLE (Matrix and Architecture Reconstruction Plug-In for Eclipse) tool is used which reconstruct software architecture and compute matrix that are used for pattern recovery. Joiner and neural network modules play key role in the recognition of design patterns. The approach has low precision and recall rates for Command, Strategy and Mediator due to difficulty in detecting these patterns.

Dynamic Analysis

Most design patterns have not only structural but also significant behavioral aspects. The dynamic analysis approaches cover areas such as performance optimization, software execution visualization, behavioral recovery and feature to code assignment. These approaches capture system behavior, but it is not practical to verify the logic of a program. It complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. The objective of dynamic analysis is to check the dependency between the classes when they delegate responsibilities to other classes during run time through calling methods of other classes. They focus on the sequence of some actions, which are performed run times by executing the program.

In [11], behavioral analysis is performed on Java.Awt package and JCL libraries. The UML sequence diagrams are used as rules for characteristics of different patterns. The approach has discovered only Bridge, Strategy, and Composite patterns. The applied technique complicates the search by expanding the set of candidate classes and results in analyzing more unrelated execution traces. We think that structural analysis should be used with the behavioral analysis to narrow down the search space for detecting patterns.

Reference [42] proposed behavior driven design pattern recovery technique which consists of a feature oriented dynamic analysis and two phase design pattern detection process. The dynamic analysis operates on the system's scenario-driven execution traces and produces a mapping between features and their implementation at class level. They applied approximate matching and structural matching algorithms using PDL and have conducted experiments on three different versions of JHotDraw (v 5.1, v6.0b1, v7.1). Their approach recovered Adapter, Proxy, Observer, Decorator, Bridge, Strategy and State.

Meyer et al. [43] have presented a technique for detecting design patterns by combining static and dynamic analysis. The approach transforms the behavioral aspects of a design pattern into finite automate and then identify the relevant method calls. The relevant calls are monitored during run time and matched against the automata to recognize patterns from the source code. The approach implemented dynamic analysis partially and the transformation of behavioral pattern into DFAs is performed manually. State and Strategy patterns are only implemented. The experiments are performed on Eclipse Framework [44].

The approach presented in [25] used static and dynamic analysis to extract program intent in order to recognize patterns from Java source code. The technique has reclassified all the GoF patterns in context of reverse engineering. The tool PINOT [98] is used to perform experiments on different open source systems. The presented approach and tool claim to recover patterns by taking information from AST and obligatory definition of design patterns. The tool recovers all the GoF [13] patterns from the source code with the exception of Prototype, Iterator and Builder patterns. The tool also generates summary of extracted patterns and their participating classes, but it does not mention any precision and recall of extracted results. Furthermore, we found large number of false positives which are recognized as true positives.

Gueheneuc et al. [45] presented a multilayered semiautomatic approach for design pattern detection from Java source code examples. The approach uses static analysis to detect static relationships. The dynamic analysis uses trace analysis techniques to compute exclusivity and life time relationships for aggregation and composition relationships. The approach is based on three layers. The first layer is used to extract different models from the source code such as metamodel, etc. The metamodel consists of classes, interfaces, methods, data types, etc. The second layer extracts idioms from metamodel and specifies relationships between different classes and other artifacts. The third layer is used to identify the design patterns from the abstract model representing the source code by using explanation-based constraint programming and constraint relaxation. Experiments are performed on five open source systems (JHotDraw v5.1, JRefactory v2.6.34, JUnit v3.7, MapperXML v1.9.7, QuickUML 2001) and they obtained an average precision of 34% for 12 design motifs. They also performed experiments on 33 industrial case studies to extract different design motifs. The approach also ensures traceability of design motifs between the implementation and the design artifacts. Authors do not consider semantic analysis to extract the intentions of the developers, which is important for recovery of patterns which have similar structural and behavioral aspects. Authors of same group presented an extended approach [66] using constraints programming supplemented with numerical analysis. They recovered patterns shown in Table 3.2 using same examples with improved precision.

An approached based on data mining and a neural network is presented by [46] to detect behavioral design patterns from different Java source code examples (ADTree, BayesNet, Prism, etc). They claim average precision of 76% on all behavioral design patterns. The MARPLE tool is used to perform experiments. The experiments are performed on toy examples, which cannot ensure the external validity and the scalability of approach.

Arcelli et al. [47] extended their work and presented a dynamic analysis approach for design pattern detection as another attempt. The technique tries to identify set of rules which contain information to identify the design pattern instances. Rules are assigned

weights to determine the importance of the design pattern for their recognition. The behavior of each design pattern is defined by combination of different rules. The rules are customizable and independent of any programming language. JADEPT prototyping tool is used to analyze its own structure, which is composed of 151 classes and it recovered Chain of Responsibility, Observer and Visitor patterns with accuracy of 100%, 90% and 17% respectively.

The approach presented in [48] has used static and dynamic analysis for detection of design patterns. They classified design patterns based on their structural, behavioral and implementation based features. The experiments are conducted on very small examples, which are developed locally and not available publicly to validate the results of approach. The comparisons with the other tools also look only descriptive.

Static Program Analysis

Most design patterns have both structural and behavioral aspects. The pure structural analysis approaches extract patterns on the basis of structural relationships between classes, but their results report high rate of false positives and false negatives. Most of structure analysis techniques are supplemented with behavioral or semantic analysis to extract patterns with improved precision and recall. The behavioral analysis approaches take input from static analysis to reduce the false positive rate for pattern recovery. Neither structural nor behavioral analysis is successful to recover all the patterns. Static program analysis techniques extract behavioral information from method bodies represented in AST/Source code. The object creation and returning by methods can be recognized by using static program analysis, which is required for Factory method, Builder, Prototype and Abstract factory patterns.

3.1.3 Semantic Analysis Approaches

Some patterns are similar in structural and behavioral aspects but they only differ in intent for which they are used. Strategy, State and Bridge are examples of such patterns. Different techniques are used for semantic analysis. Reference [35] proposed the following guidelines for semantic analysis:

1. Searching design documentation of the source code.
2. Searching in-line comments in the source code.
3. Getting clues from naming conventions of classes.

Option 1 is rarely useful because the information related with documents is not consistent with the source code and sometimes documents are not available. The in-line comments give only information about the intent of source code and give no information related with patterns. The availability of comments in source code of different systems is also sometimes vague and not useful due to revisions and maintenance in the source code. The approach preferred naming conventions for semantic analysis and they showed some

successful results in their extraction process. It also depends on the disciplines of programmers that how they have used proper naming conventions in the source code.

Paschke [49] has presented a semantic design pattern language for complex event processing. The major focus of the approach is to provide a formal, machine readable pattern definition language, which is flexible enough, human readable and provide meta-data annotations that allow reasoning for patterns. The approach is implemented as semi-formal XML pattern description language using machine-processable XML markup, human-readable narratives and formal ontology design language (DOL). The approach is based on theoretical background and is not applied to the recovery of patterns.

Alnusair et al. [50] have presented a model driven approach for recovering design patterns from the source code of different examples. They argue that an effective pattern recovery approach requires semantic reasoning to properly match an ontological representation of both: conceptual source code knowledge and design pattern descriptions. They represented conceptual knowledge available in the source code in SCRO (Source code representation Ontology). SCRO's knowledge is represented using the OWL-DL ontology language and verified using the Pellet OWL-DL reasoner [51]. They have experimented on Singleton, Visitor, Composite and Observer using JHotDraw 6.0b1, JUnit 3.7 and Java AWT 1.6.

3.1.4 Approaches Considering Pattern Compositions

A number of approaches focused on the formalization and composition of design patterns to supplement different pattern detection approaches by formally specifying patterns. Formal specification languages are used to specify design patterns. Some design pattern detection techniques use pattern specifications of other approaches in their implementation to detect patterns from the source code. Most of the specification languages have tool support to validate the specifications for correctness and completeness. Traditional UML specifications and textual descriptions cannot capture the essence and the intent of patterns. Design patterns have different implementation variants and formal specification of patterns can help to specify the possible variations in different patterns. An overview about formal specification techniques is given in Subsection 2.3.1. This section discusses approaches which have used formal specifications for design pattern detection or the formal specifications suggested by these approaches can be used for pattern detection by other approaches. The composition of design patterns can be recognized easily if patterns are formally specified.

Wang et al. [52] have presented a generic mathematical model of design pattern specifications using Real Time Process Algebra (RPTA). They specified Strategy, State and Master slave patterns by their approach. The approach uses 34 notations to denote class association relationships and specify patterns from three facets known as the architectures, static behaviors and dynamic behaviors of patterns. They claim that their

generic model in not only applicable to existing pattern description and comprehension, but it is also useful for future pattern identification and formalization.

The approach presented in [53] has focused on the composition of design patterns. A concept of composition of patterns with respect to overlap is formally defined based on specialization and lifting operations on patterns. The approach provides very good theoretical background for the composition and the formal verification of design patterns. Pattern recovery tools can use presented formal definitions (Composition, Overlapping) to detect composition and overlapping between design patterns.

Bayley et al. [54] presented an approach that deploys predicate logic to specify conditions on the class diagrams that describe design patterns. The approach has specified all GoF patterns and also shows the relationships between different patterns. They use intersection operation to check whether two patterns can be composed. The formal notations and descriptions used in the approach are very expressive and understandable for a novice user. The authors extended their approach in [55] to specify the variants of design patterns and clarify the ambiguities of informal descriptions. The major focus of the extended approach is on expressiveness, human readability, adequacy, formal reasoning, transformations and tool support for the pattern specifications. The approach has specified static and dynamic features of each pattern as static and dynamic conditions separately.

Eden et al. [56] presented LePUS, a formal specification language to specify all GoF patterns. The approach models class relationships, semantics, facilitates reasoning and improves the reasoning with higher order set. The program is represented as set of major entities and relationships among them. The approach has advantage of using higher order logic, but it is difficult to map the LePUS specifications into executable program.

Dania et al. [69] have proposed an ontology based approach for the integration of design patterns. They focus on extracting the intent of a design pattern and how to replace a design pattern with alternative model. Their approach is based on model driven processes which contain the expert knowledge expressed in terms of patterns. The OWL language is used to specify this knowledge in the form of ontology. SPARQL is used to extract relationships from the data represented in RDF graph. The relationships between Composite and Decorator patterns are analyzed.

3.2 Review of Design Pattern Recovery Tools

Automated design pattern recovery techniques are assisted by different tools to match different instances of patterns from the source code of legacy applications. The capability of a pattern matching tool depends on several factors including, extraction, abstraction, presentation, scalability and accuracy. Most of the approaches apply their custom build tools, which are developed to validate concepts of methodologies used for pattern

recovery, but these tools are not available publicly for the validation of their results. Some approaches used third party tools for the intermediate representation of the source code and then use plug-In techniques to support their recovery process. The accuracy of these tools depends on the extraction capability of third party tools. Another type of approaches represent source code in the form of abstract syntax tree by using parsers of different languages and then apply different techniques to extract patterns from the source code. We have noticed that most of the tools are language dependent and they support only (C/C++, Java and Smalltalk) languages for pattern recovery. The tools also differ in input and output formats which make their compatibility difficult or impossible with other tools. Furthermore, little attention is paid toward the development of open source tools for design pattern recovery. The important tools used in different approaches are mentioned in Table 3.3. These tools are selected for comparison based on their results extracted with good precision, recall, available documentation and extracted pattern instances. An overview and comparison about the features of important tools used for pattern recovery is given below:

3.2.1 DPRE (Design Pattern Recovery Environment)

DPRE is a prototyping tool which is generated from a grammar specification by using visual language grammar VLDesk (Visual Language Desk) system [57]. The main components of the VLDesk architecture are: the Symbol Editor, the Visual Production Editor, the Textual Production Editor and the Visual Programming Environment Generator. DPRE is a general purpose tool and it supports different reengineering activities. It includes a UML class diagram editor and visualizes imported class diagrams. Indeed, DPRE supports SVG as internal data format, and includes a visualization graph algorithm to visualize class diagrams. It recovers structural design patterns such as Adapter, Bridge, Composite, Decorator, Façade and Proxy from different versions of JHotDraw, JUnit, JRefactory, etc. DPRE is implemented as Eclipse Plug-In. The tool is used by the approaches [19 29] as discussed in the Subsection 3.1.1. It suffers the problem of scalability and disparity of results is noticed as discussed in Section 3.3. The precision and recall of tool is mentioned in Table 3.1. The tool is available for download on the web (http://www.sesa.dmi.unisa.it/dpr).

3.2.2 Columbus

Columbus is a general reverse engineering framework that is used to handle number of reverse engineering tasks. It supports project handling, data extraction, data storage, data visualization and filtering. The framework has been developed in co-operation between the research group on Artificial Intelligence at the University of Szeged, the Software Technology Laboratory of the Nokia Research Center and the FrontEndART Ltd [58]. The basic architecture of Columbus consists of different plug-ins namely: extractor, linker

and exporter. The architecture of Columbus allows user to add new plug-in to Columbus system using the plug-in API. Columbus is used to handle general reverse engineering tasks with partial support for design pattern recovery. The performance of framework degrades with increasing size of the software. It is available freely for scientific and educational research. Columbus uses the graph matching technique to detect patterns from the source code. It is used to extract only Adapter, Strategy and State patterns from C++ source code examples. The tool shows precision of 67-95% on StarWriter [136]. Columbus is used as framework by different reverse engineering tools and is available on web (http://www.univie.ac.at/columbus/documentation/documentation_main.html).

3.2.3 DPVK (Design Pattern Verification Toolkit)

It is a toolkit that is used to detect design patterns from Eiffel programs. It uses groke, a text search tool for extracting static relationships among different classes. In terms of static structure, DPVK takes inheritance and method invocation relationships between classes into account. In DPVK, each design pattern has two definitions: one definition is based on the static structure of the pattern and the other is based on its dynamic behavior. DPVK uses special structure and behavior to identify and pinpoint design patterns and differentiate one design pattern from others. DPVK includes three modules and runs in four stages. These modules/stages are static fact extraction, candidate instance discovery and false positive elimination. The fourth stage is the manual evaluation of the results extracted by previous states. Recall and precision are 100% and 19.9% respectively. Since creational patterns have comparatively simple static structures and less dynamic interactions, their false positives are more common in the test of tool. The tool is applied on five GoF [13] patterns namely: Prototype, Singleton, Facade, Chain of responsibility, and Template method. Since the implementation of the Visitor pattern is rather complicated, the results indicate a larger number of false positives in implementation of this pattern. Currently, DPVK does not take method signatures and class types into consideration during pattern detection process. Authors of DPVK mention that its implementation is in progress and no information about tool is available on web for validation of its results.

3.2.4 DeMIMA (Design Motif Identification Multilayered Approach)

DeMIMA is a prototyping tool implemented on top of PTIDEJ framework using Java programming language. It takes benefits from existing libraries of PTIDEJ (PADL, PADL CLASSFILE CREATOR, RELATIONSHIP STATIC ANALYSER, CAFFEINE, etc.) to implement the constraint resolver. It is used as information retrieval system with major focus on the improvement of precision and recall. It detects design motifs by highlighting microarchitectures, which are helpful for the comprehension and other reengineering activities. The explanation-based constraints programming is used to identify

microarchitecutres for the maintainers to direct their search and discriminate between the false positives. The tool is tested on 33 industrial components as well as open source systems with average precision of 43.2%, 25.9%, 43.8%, 22.4%, and 34.8% on (JHotDraw v5.1, JRefactory v2.6.24, Junit v3.7, MapperXML v1.9.7 and QuickUML 2001) for different design patterns. DeMIMA is extensible, scalable and ensures traceability between the implementation and design artifacts. The detailed information about tool is available on web (http://www.ptidej.net/research/demima/).

3.2.5 DPD (Design Pattern Detection)

DPD is implemented in Java and it employ java byte code manipulation framework, which extracts the static structure of the legacy system including abstraction, inheritance, constructor signatures, method signatures, method invocations and object instantiation. This primary information is used to extract more advanced properties of design patterns like collection element type checking, similar abstract method invocation, abstract method adaption and static self reference. Lastly, the extracted information is used to generate the matrix that describes the system to be examined. In the current implementation, pattern descriptions are hard-coded within the program. The recall rate for patterns (Adapter, Composite, Decorator, Observer, Prototype, Singleton, Template method and Visitor) is 100 %. The recall rate for Factory method is 66.7 %(JHotDraw v5.1), 25 %(JRefactory v2.6.24) and 100% (JUnit v3.7). Similarly, the recall rate for State/Strategy is 95%-100%. DPD extracts fast results on medium size examples, but it hangs up on large software systems. It is available on web (http://java.uom.gr/~nikos/pattern-detection.html) for validation.

3.2.6 PTIDEJ (Pattern Traces Identification, Detection, and Enhancement in Java)

The PTIDEJ system is an automated open source system implemented in Java which uses approximate and exact matches to extract different idioms, micro-architectures, design patterns and design defects from the source code. The layered architecture used in the implementation of PTIDEJ provides flexibility for its extension. It uses PADL meta-model (Pattern and Abstract level Description Language) for describing models of program. It has parsers for representing metamodel of AOL, C++ and Java. It is a tool suite, which is generic and used for the analysis and maintenance of object-oriented architectures. It includes four tools: (1) SAD, a tool for the detection and correction of software architecture defects; (2) EPI, a tool for the efficient identification of design patterns occurrences in a program; (3) DRAM, a tool for the visualization of the static and dynamic data of programs with adjacency matrices; and (4) Aspects, a tool for modeling and computing metrics on aspect-oriented abstractions. The experiments are performed on small test cases, HotJava v3.0 (90 classes), JEdit v3.1 (250 classes), JHotDraw v5.1 (155 classes) and JUnit v3.2 (90 classes). PTIDEJ is available on web (http://ptidej.iro.umontreal.ca/) for download and application.

3.2.7 PINOT (Pattern Inference and Recovery Tool)

PINOT is a fully automated design pattern recovery tool. It is modification of Jike (an open source Java compiler written in C++). It is a command line tool used to detect all the GoF patterns except Prototype, Builder and Iterator. PINOT reduces the search space for detecting patterns by selecting the properties of patterns playing key role in an order. The tool only analyzes lazy instantiation that uses boolean or java.lang.Object types, but it is not able to detect the eager instantiation and the placeholders for Singleton. Inter-procedural data-flow and alias analyses are only used for detecting patterns that often involve method delegations in practice, such as Abstract factory method, Factory method, Strategy and State patterns. PINOT cannot recognize any user defined or user-extended data structures. It is not customizable because it uses algorithms which are hard coded in the source code. It presents its results in a report, which shows the path of source code classes playing roles in each pattern. Precision and recall is not mentioned by authors which is important for comparisons with other tools. It is available on web (http://www.cs.ucdavis.edu/~shini/research/pinot/) for experimentation.

3.2.8 DP-Miner

DP-Miner is intended to recover design patterns applied in software systems. Tool analyzes structural, behavioral, and semantic characteristics of system and patterns. In structural analysis, DP-Miner presents system structure in a matrix with columns and rows to be all classes in the system. The value of each cell represents the relationships among the classes. The structure of each design pattern is similarly represented in another matrix. The discovery of design patterns from source code becomes matching between the two matrices. If the pattern matrix matches the system matrix, a candidate instance of the pattern is found. Besides matrix, DP-Miner uses weight to represent the attributes/operations of each class and its relationships with other classes and compares system class weight to pattern class weight. Tool is available as standalone version which detected only four patterns (Adapter, Composite, Strategy and State) and Eclipse Plug-In version covers broader range of patterns. Precision and recall are mentioned in Table 3.1. Both versions of DP-Miner are available on web (http://www.utdallas.edu/~xyz045100 /DesignPattern/DP_Miner/index.html) for validation.

3.2.9 D^3 (Detection of Diverse Design Patterns)

D-cubed is a research prototype which utilizes structural and behavioral information for pattern recovery. It is a command line tool written in Java. Tool takes information from the program metamodel and executes the SQL queries to match patterns with the source code. It computes the transitive closure using DFS for the structural and the call flow analysis for behavioral aspects. It is used to detect Singleton, Factory Method, Abstract

Factory and Builder patterns from the source code of Java examples. Tool recovers some variants of structural design patterns which are not detected by PINOT and FUJABA, but it does not mention any precision and recall. The extracted results do not mention location of patterns in the source code. Authors mentioned that tool will be extended to extract behavioral design patterns. D-Cubed is not available publicly to validate its results.

3.2.10 SPQR (System for Pattern Query and Recognition)

It is a toolset used for detection of elemental design patterns (EDPs) and GoF [13] patterns from C++ source code examples. Single scripts provide workflow around several tasks such as source code feature detection, feature rule description, rule inference and query reporting. In order to perform such activities, the structure of tool consists of various tools (gcctree2poml, poml2otter OTTER and proof2pattern) to perform different tasks. The initial objective of SPQR was to detect design patterns which are recently extended to architectural patterns. Each tool performs its task independently and can be used for other languages as well. The experiments focused on the EDPs and only Decorator and Singleton are discussed on different examples. The detailed information about SPQR is available on the web (http://www.cs.unc.edu/~smithja/SPQR.html).

Table 3.3: Comparison of features of different Design Pattern Recovery tools

Tool	DPRE	DeMIMA	DPD	Columbus	DPVK	Ptidej	PINOT	DP-Miner	D3	SPQR
Supported platforms	WN	WN	WN	WN	MP	WN	Linux	WN	NM	MacOS
Supported Languages	C++ Java	Java	Java	C++	Eiffel	Java	Java	Java	Java	C++
Founder	Uni of Salerno	Uni of Montréal	Uni of Macedonia	Nokia + FrontEndART Ltd	Uni of York	Uni of Montréal	Uni of California	Uni of Taxas	Uni of Warsaw	Uni of North Carolina
Input Format	SVG VLG	-	Matrix	RSF GXL	RSF	PADL	AST	XMI	AST	AST, POML
Output Format	UML	UML	Text	UML	XML	Text	Text	Text	NA	UML
Space Occupied	685 KB	NA	424KB	NA	NA	25.79 MB	NA	832KB	NA	NA
Availability	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes
Accuracy	Good	Good	VG	Good	Good	Good	Med	VG	Good	Good
User interface	Yes	Yes	Yes	Yes	No	Yes	No	Yes	NA	Yes
Visualization	Yes	Yes	No	Yes	No	Yes	No	Yes	NA	No
Type	Open	RP	RP	open	RP	open	Com	RP	RP	RP
Scalability	Good	VG	Good	Good	NA	Good	Good	VG	Med	good
Portability	Yes	No	No	Yes	No	Yes	No	Partial	No	Yes
User defined features	No	No	Yes	No	No	No	No	No	Yes	Yes

MP: Multiple Platforms WN: Windows RP: Research Prototype Com: Commercial NA: Not Available

3.3 Comparisons of Results

It is necessary to collect, organize and analyze several sources of information related with different design patterns recovery approaches and tools that can be used for evaluating and comparing results. The key factors for evaluation are search criteria, precision/recall, benchmark systems, documentation examined of source code, etc. We focused on comparing results of different approaches based on precision, recall and F-Score extracted by different approaches. See Section 8.4 for detail about F-Score. A wide disparity is analyzed in the results of different approaches, which motivated us to explore the causes of disparity. The major causes for the disparity of results from different approaches are different definitions of the same pattern, approximate/exact matches, search criteria, implementation variants, etc. Some approaches take standard GoF definition of each patterns and they do not consider different implementation variants of same pattern. To the best of our knowledge, no common definition exists for each design pattern which

causes the disparity of results from different approaches. Thus wide disparity in results extracted by different approaches set some open issues that reflect the attention of design pattern and the reverse engineering community toward automated design pattern recovery. Reference [1] mentioned the causes of disparity in results by different approaches and suggested benchmark systems for the validity of results. Different groups are working on the benchmark systems which will be used to compare the results of different approaches. To date, only benchmarks for few systems are available which include only few patterns. Through deep insight view of different approaches, we found inconsistencies in the results of different approaches as shown in Tables 3.4 and 3.5. The wide disparity is noticed in the results of [20 45 19] on Adapter pattern in JHotDraw 5.1. The authors detected 18, 1 and 41 instances of Adaptor, which reflect wide disparity in the results of these approaches. There is only one common instance detected by all approaches and it is also difficult to realize that common instance, because each approach presents its results in different formats. Some approaches only show the number of extracted patterns and it becomes difficult to realize the extract location of each pattern in such approaches. Similarly, Table 3.5 shows the results of Bridge pattern extracted by [19 25 59] from JHotDraw 6.0b1. The approaches have extracted 166, 107 and 58 instances respectively. It reflects that only 58 instances are commonly extracted by all approaches. The dilemma still remains on the realization of common instances. Similarly, the approach presented in [19] extracted Adapter pattern from JHotDraw 6.0b1 with AbstractFigure, NullFigure and HandleEnumeration playing roles of Target, Adapter and Adaptee classes. The handle method in the NullFigure which delegate request is missing in the AbstracFigure. Moreover, AbstractFigure is a concrete class which extends to another class.

Table 3.4: Comparison of results recovered by different approaches on same software

Software	JHotDraw 5.1			Junit 3.7			JRefactory 2.6.24			NETBEANS 1.0.x	
Reference	**[20]**	**[45]**	**[19]**	**[20]**	**[45]**	**[50]**	**[20]**	**[45]**	**[60]**	**[25]**	**[60]**
Singleton	2	2	x	0	0	x	12	2	1	0	2
Adapter	18	1	41	1	0	x	7	17	16	8	1
Composite	1	1	0	1	1	1	0	0	x	0	0
Decorator	3	1	0	1	1	x	1	0	x	0	0
Factory Method	3	3	x	0	0	x	4	1	0	0	18
Observer	5	2	x	4	3	4	0	0	x	0	0
Prototype	1	2	x	0	0		0	0	x	0	0
Command	0	1	x	0	2	0	0	0	0	1	8
Template Method	5	2	x	1	0	1	17	0	x	0	0
Visitor	1	0	x	0	0	0	2	2	2	0	0
State	23	2	x	3	0	x	12	2	3	0	0

X: approach is not applied

We tested standalone version of DP-Miner [143] tool on Junit 3.8.2 with the XMI file that is available on the website of tool. The tool detects "0" instance for the Adapter and Composite design patterns, while authors mention "3" instances of Adapter and Composite in their approach presented in [59]. The tool also takes a lot of time on loading XML file from the source directory and does not give any information about the exact path on which detected instances of patterns are located. The results of such approaches cannot be used for the further analysis of pattern's results. Similar disparities are also noticed in the results of other approaches which are highlighted in our result evaluation section mentioned in Chapter 8.

During analysis of NETBEANS, the approaches [25 60] differ in their results in the case of Adapter, Factory method, Singleton and Command patterns. Similar disparities are analyzed in results of JRefactory 2.6.24 extracted by [20 45 60] in the case of most patterns as mentioned in Table 3.4. For example, in the case of State pattern, each approach has extracted 12, 2 and 3 instances respectively. Another variation in results is clear from Table 3.5 for Bridge pattern. The approaches [19 25 59] have extracted 166, 107 and 58 instances of Bridge pattern respectively.

Table 3.5: Comparison of results recovered by different approaches

System	JHotDraw6.01			Apache Ant 1.6.2			Java Swing 1.4		Quick UML 2001			Java AWT(1 2 3)		
Reference	**[19]**	**[25]**	**[59]**	**[19]**	**[25]**	**[20]**	**[19]**	**[25]**	**[19]**	**[45]**	**[20]**	**[25]**	**[59]**	**[50]**
Adapter	53	5	4	71	13	4	159	17	27	0	11	3	21	x
Bridge	166	107	58	51	5	x	353	142	22	x	x	15	65	x
Composite	5	4	4	5	44	14	4	20	0	3	1	3	3	3
Facade	20	94	x	111	79	x	145	101	16	x	x	58	x	x
Proxy	0	17	x	0	27	4	2	43	1	x	6	13	x	x
Decorator	0	5	x	0	4	14	0	15	0	0	0	3	x	x
Strategy	x	51	64	x	19	x	X	96	x	0	15	54	76	x

Java AWT (1)= Java AWT 1.3: Java AWT(2)= Java AWT 1.4: Java AWT(3)= Java AWT 1.6 x: approach is not applied

Finally, we analyzed the pattern recovery results extracted by [29 61 28] on different systems implemented in C++. The disparity is noticed in the results of [29 61] for extracting Adapter pattern from Galb++ [62]. Similarly, the results of both approaches differ for extracting Bridge pattern from Libg++ [63]. The both approaches have similar results in the case of most patterns because the size of systems under examination is small. We also notice that disparity of results increases as the size of software used for pattern extraction increases.

The major reasons for the disparity of results are the different design pattern definitions adopted by each approach, approximate/exact matching mechanism, different variants and roles of classes participating in the recognition of patterns.

Table 3.6: Comparison of results from C++ applications

System	Galb++ 2.4		Libg++ 2.7.2		Mec 0.3		Socket 1.10		LEDA 3.4	
Language	C++		C++		C++		C++		C++	
Reference	[29]	[61]	[29]	[61]	[29]	[61]	[29]	[61]	[61]	[28]
LOC	20,507	20,507	44.106	40,119	21,006	21,006	3078	3078	115,882	85 606
Classes	55	55	144	167	32	32	30	29	208	1617
Adapter	6	4	0	0	1	1	1	1	2	3
Bridge	0	0	1	3	0	0	0	1	0	0
Composite	0	0	0	0	0	0	0	0	0	0
Proxy	0	0	0	0	0	0	0	0	0	0
Decorator	0	0	12	12	0	0	0	0	0	0

3.4 Critical Review and Observations

With the deep analysis and the review of different up-to-date works in the area of design pattern recovery, we have following observations. These observations are used to setup objectives for our approach.

i) Most approaches target open source systems during pattern recovery, which do not have proper documentation. It is very difficult to compare results of each approach because wide disparity exists between the recovered results of different approaches.

ii) Most approaches perform experiments only on source code of C/C++ or Java languages for pattern recovery. Although, the legacy applications are developed in these languages, but some commercial and business applications which are developed using patterns in other languages are not paid any attention.

iii) Some approaches recover only few patterns with good precision and recall, but the real applications are developed using larger number of patterns. So the applications of these approaches are not worthwhile for legacy systems, which are developed by using broader range of patterns. On the other hand, some approaches claim to recover the broader range of patterns, but their recovered results have high ratio of false positives and false negatives as seen in Tables 3.4, 3.5 and 3.6.

iv) The recovered pattern results are very important for program comprehension. A number of pattern recovery approaches give information about the number of patterns they have discovered, but they do not give any information about the exact location of these patterns in the source code. So the extracted information cannot be used for comprehension, maintenance and other reengineering activities.

v) Most approaches focus only on GoF or subset of GoF patterns. Specifically, approaches based on hard coded algorithms cannot be extended to recovery of J2EE patterns, architectural patterns and Web based patterns.

vi) In real applications, smaller patterns are composed to build the larger patterns. There are some good attempts on the formal composition of design patterns, but most of the pattern detection approaches overlook the detection of composition and overlapping of design patterns.

vii) Most approaches inculcate structural and behavioral analysis for pattern recovery with the exception of only few approaches, which used semantic analysis for their recovery process. We think that semantic analysis is very important because some patterns have similar structural and behavioral properties and they cannot be truly recognized without semantic analysis.

viii) Each approach takes his own pattern definition and there is no agreed upon solution for the whole community to validate the same definition and interpretation. Further research and cooperative work is required to solve this problem.

ix) Most approaches show that the results of structural and behavioral analysis of State and Strategy pattern are identical because they have same structural and behavioral properties, but these approaches do not care about different intents during pattern recovery.

x) Most approaches performed experiments on one or two examples and they do not cross validate their results with other approaches.

3.5 Requirements for Design Pattern Recognition Approach

A comprehensive review of existing approaches listed in Table 3.1 gave us insight view of approaches used for design pattern recovery in the past. We derived the observations and findings from problems mentioned in Section 2.4 and related work presented in this chapter. These observations and problems are the challenges in the area of automated pattern recovery which become our requirements. The central goal of design pattern recovery techniques is to detect patterns from legacy source code accurately which ultimately help program maintenance, source code refactoring, program understanding, reverse engineering and reengineering existing applications. In order to achieve this goal, we focus on the following requirements as contribution of this dissertation:

Requirement 1(Recognize Variants of Design Patterns)

The structural and implementation variants of design patterns challenge pattern recovery approaches as discussed in Section 2.4. The variation problem is due to different interpretations of the same pattern and is highlighted by [1 10 37] as requirement for design pattern recovery.

Requirement 2(Customizable Pattern Definitions)
This requirement became obvious for solution of requirement 1, because the approaches based on hard coded algorithms do not let user to refine pattern definitions. The reusable and customizable pattern definitions can handle pattern variations.

Requirement 3(High Precision and Recall)
Most approaches use single recognition technique and their extracted precision and recall is low due to not filtering false positives and missing false negatives. In the literature, the approaches like [23 47 53] used the concept of only SQL for pattern detection. The SQL are not capable to extract all patterns with the same accuracy.

Requirement 4(Multiple Language Pattern Detection)
The review of Table 3.1 reflected that most of approaches are specific to single language for pattern detection and they cannot be generalized for multiple languages. The multiple language pattern recovery became requirement for this dissertation.

Requirement 5(Evaluation of the Approach)
Each design pattern detection approach has to be evaluated concerning implication of its results and the effort for customizing pattern definitions in order to detect the variants of different patterns. The approach can be adopted by the academia/industry if its empirical evaluation clearly elaborates benefits of approach in comparison with existing work. We analyzed disparity in the results of different approaches in Section 3.3. The causes of disparity become obvious requirement for cross validating and comparing results of our approach with different approaches. This can lead to trusted benchmark examples as suggested by different authors [1 35 67].

3.6 Summary

This chapter presented review about different pattern recovery approaches and tools. Section 3.1 discussed the state of art used by different approaches with their strengths and limitations. The state of art techniques extract design pattern instances based on structural analysis, dynamic analysis and semantic analysis. The techniques which extract patterns based on structural analysis have very low precision and recall rates. These techniques are not able to detect all types of GoF patterns. Our technique is different from structural analysis techniques [3 19 20 22 23 29 31 32 34 36], because we used multiple analysis methods and integrated multiple searching techniques (SQL, Source code parsers, Regular expressions) for extraction of patterns. Secondly, most of the structural analysis based techniques focuses on structural design patterns, but we apply our approach on all types of GoF patterns. Thirdly, all presented approaches focused only on Java/C++ language for design pattern detection, but our approach is capable to extract patterns from source code of multiple languages. Currently, we applied our approach on all types of GoF patterns

and source code of (C/C++, Java, C#), but it can be extended for more than ten languages which have reverse engineering support from Enterprise Architect Modeling tool [26]. Dynamic analysis supplements the structural analysis approaches to improve the precision and recall rates. The major problem with dynamic analysis approaches is to filter out unrelated execution traces. Most of these techniques take data from structural analysis techniques and then apply behavioral analysis on the extracted static facts in order to reduce the search space. We extracted dynamic information from the source code using source code scanners and parsers for each language. Source code parsers use the static facts extracted from structural analysis phase and extract (delegations, aggregations, method calls, etc.) information directly from source code. The combination of structural, dynamic and semantic analysis is used only by [35]. During analysis of different open source systems, we observed that proper naming conventions are seldom used and recognition of consistency for naming conventions is another key issue. The approach itself discusses the limitation of using naming conventions, because all the classes related with any pattern may not have proper naming conventions. Further, studies [138 139] confirm that source code modifications rename source code elements which affect the naming conventions. We extended the concept of using annotations for semantic analysis in comparison with using naming conventions as discussed in Subsection 6.2.4. Section 3.2 highlighted the role of different tools which supplement different pattern recovery approaches. The selection of tools is based on their availability, successful application in recent recovery approaches and good precision/recall rates. Section 3.3 elaborated disparity of results by different approaches. The major cause of disparity in the results is due to different interpretations of design patterns by different authors. Section 3.4 discussed our observations related with recent recovery techniques and tools which become motivation for our approach. Finally, the requirements which improve accuracy of design pattern detection, based on the observations and problems related with existing approaches are discussed in Section 3.5.

Chapter 4

Overview of Pattern Recognition Approach

The concept for the approach proposed in this thesis stems from last two chapters, which discuss the problems related with documentation, specification, different pattern recognition techniques and tools. The requirements for defining customizable feature types and pattern definitions arise from Section 2.4 and the problems related with current recovery approaches discussed in Section 3.4. This chapter briefly presents the main concept of approach and the following two chapters describe in detail its two main phases.

Section 4.1 discusses the idea of approach based on requirements and problems. Section 4.2 discusses the concept of feature types and pattern definitions. The concept of multiple search techniques based on feature type requirements is discussed in Section 4.3. Finally, Section 4.4 discusses the major challenges faced by the approach and the concepts that have been used to handle these challenges.

4.1 Main Concepts of the Approach

The main concept of approach used in this thesis is based on identified feature types which describe features of patterns that are customizable by the user and can be mapped to pattern definitions. The feature types are used to define variant pattern definitions which are used to recognize patterns. The recognition quality of pattern detection approach is dependent on accurate, adaptable, reusable and variant pattern definitions. The numbers of presented approaches use different definitions of patterns which cause disparity in the results rendered by majority of pattern recognition approaches. We have major focus on variant pattern definitions by using instantiated feature types. The concepts of alternative and negatives feature types are used to filter out wrong matches and improve accuracy of pattern recognition process. Each feature type requires a single or a combination of recognition technologies such as SQL queries, Regular Expressions and Source code parsers with the help of static and dynamic parameters. Recognition queries are used to translate feature types for matching definitions of patterns. The recognition queries use static and dynamic parameters to link different feature types. The focus of approach is to define features types which are comprehensible, customizable, variable and can map to variant pattern definitions as discussed in the next section. Pattern recognition approach is divided into two phases: creating pattern definitions and pattern recognition.

4.2 Phase 1: Creating Pattern Definitions

The pattern recognition process is based on creating reliable, variant and customizable pattern definitions. The approach uses pattern definition process for creating variant design pattern definitions by using feature types. Feature types are the backbone for pattern definitions. User can select an existing feature type or can define a new feature type that suits single pattern definition. The basic question rises that how feature types for pattern definitions are selected. Figure 4.1 shows abstract relationship between feature types and design pattern definitions.

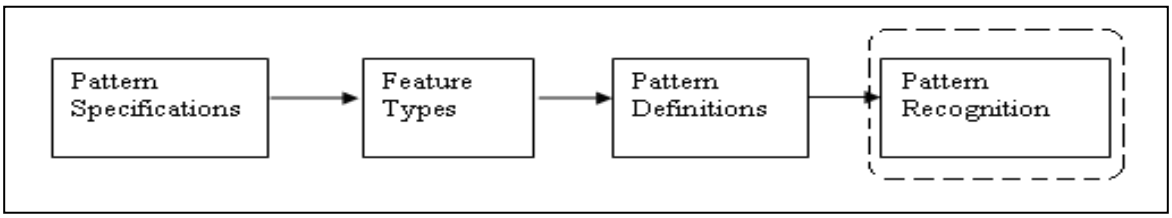


Figure 4.1: Relationship between feature types and pattern definitions

We take specification of each design pattern from the standard UML class diagram and sequence diagram of each design pattern. The specification is used to define feature types for each design pattern. The possible variants of different patterns are focused while defining feature types. The numbers of related feature types are used to define different patterns with number of variants. Further, pattern definitions are used for pattern recognition.

We explain relationship between constructs of Figure 4.1 by Singleton design pattern. The implementation variants of Singleton make it difficult to specify, define and detect Singleton by generic one definition. It is important to take alternative specification for each variant, which make it possible to recognize its all variants accurately. The possible identified variants of Singleton are (Standard Singleton, Subclass Singleton, Subclass Singleton with hashtable, Singleton counter, Singleton with different placeholders, Milton, etc.). Figure 4.2 further explains how Singleton specifications are related to feature types and pattern definitions. The variants of Singleton are mentioned on the left side of the Figure 4.2. The detail about specification of each variant is given in Subsection 2.4.1. The middle part has number of feature types which are related to one or more specifications of Singleton. The definitions for detecting different variants of Singleton are shown in the right side of Figure 4.2. The complete list of feature types for all possible variants of Singleton is curtailed in Figure 4.2 due to limitation of space. It is also noteworthy to mention that some feature types are only used as helping feature types. Helping feature types only support main feature types for matching different instances of patterns. The helping features types are not mentioned in this example and are given in Appendix C.

Similar definitions for other variants of Singleton can be created by using different feature types. Variant 2 as mentioned in Figure 4.2 is used to match the subclass Singleton. It uses the features (FT1, FT2, FT4) from variant 1 and adds new features (FT5, FTn) to match the subclass Singleton. The generic concept of structured feature types allow us to use single, alternative, supporting, parser module and regular expression features to define alternative definitions for the detection of different implementation variants.

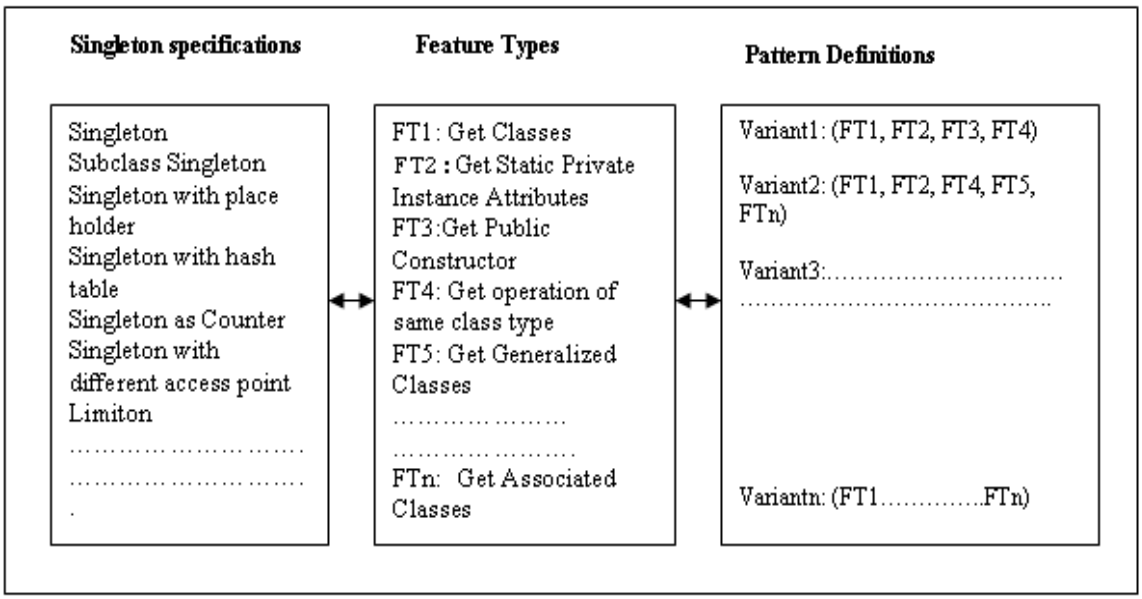


Figure 4.2: Relationship between singleton variants, feature types and pattern definitions

The concept of feature types and pattern definitions is briefly discussed in this section with example of Singleton design pattern. The process of creating pattern definitions is key to success for pattern recognition. The detail about creating pattern definitions for different patterns is discussed in Chapter 5.

4.3 Phase 2: Pattern Recognition

We used concept of integrated multiple techniques based on the following searching technologies and metadata information for pattern recognition.

I) SQL queries
II) Source code parsers
III) Regular expressions
IV) Annotations

The source code model created by Enterprise Architect Modeling Tool is analyzed using SQL queries to extract the static information related with major roles of patterns. A pattern definition is the combination of various relevant feature types which in turn use different recognition techniques. The SQL queries play key role for extracting structural

information from the source code model, which is used by the other techniques. Queries are simple to write and can extract structural information quickly from the source code model. The feature types and SQL queries are independent from programming languages. User can add and customize new features and queries without digging into the source code, because they are separate from the pattern recognition process. The queries are good option for extracting structural information, but are limited to source code model which abstract certain details of source code like aggregations, delegations, method invocations, etc. The queries in our approach also require prior step of reverse engineering which consumes time, but the use of reverse engineering tool help to present and visualize the results. Another limitation of using queries in our technique is that user requires the internal knowledge of data structure before writing queries. Finally, queries are very efficient for extracting static information like class inheritances, associations, friend relationships, interface hierarchies, modifiers of classes and methods, method parameters, method return types, attributes, data types, etc. This primary information further supplements the dynamic analysis techniques for pattern extraction.

The motivation for developing source code parser module raised due to limitation of regular expressions for matching nested information from the source code and missing capability of reverse engineering tools used to create the intermediate representations. We used the concept of source code parser module to extract information directly from the source code, which is abstract in the source code model. The parser module is developed by using CoCo/R grammar and it can be extended for different languages, because grammar for different language is freely available. It is used for the detailed analysis of source code. It can extract static as well as dynamic information related with different features of design patterns. The major focus of source code parser module is to extract (delegation, method invocation, object creation, aggregation, etc.) information directly from the source code. The information extracted by source code parser module is used in the SQL queries as parameters to extract further information from the source code model that is dependent on the source code information. This information is quite helpful for deep analysis of source code, which ultimately helped our approach to improve accuracy for pattern detection. Currently, our approach supports parser module for Java and C#. The implementation of C/C++ parser module is under progress. Parser module enriches the static information extracted by SQL, but it is specific to one searched feature type. It also requires additional implementation effort for developing parsers for each language.

The concept of regular expressions is used to extract information directly from the source code, which is not available in the source code model. Regular expressions have been used in programming and different text editors from long time for matching text. They have context independent syntax and can extract line by line information by scanning the source code. The major focus of our approach is to use customizable pattern definitions and implementation constructs, which allow user to define new features to

control implementation variants of a single design pattern. We avoided use of parser module for extracting information that can be extracted by using simple regular expression patterns. We apply regular expressions especially to extract single line expressions such as header file definitions, comments, classifiers name, annotations, etc. The regular expression patterns are very easy to write, understandable and customizable by the user. The problem with regular expressions is that they are not able to extract the nested information from the source code elements. Moreover, it is also very difficult to write language independent patterns using regular expressions. Figure 4.3 gives a high level overview about pattern recognition process.

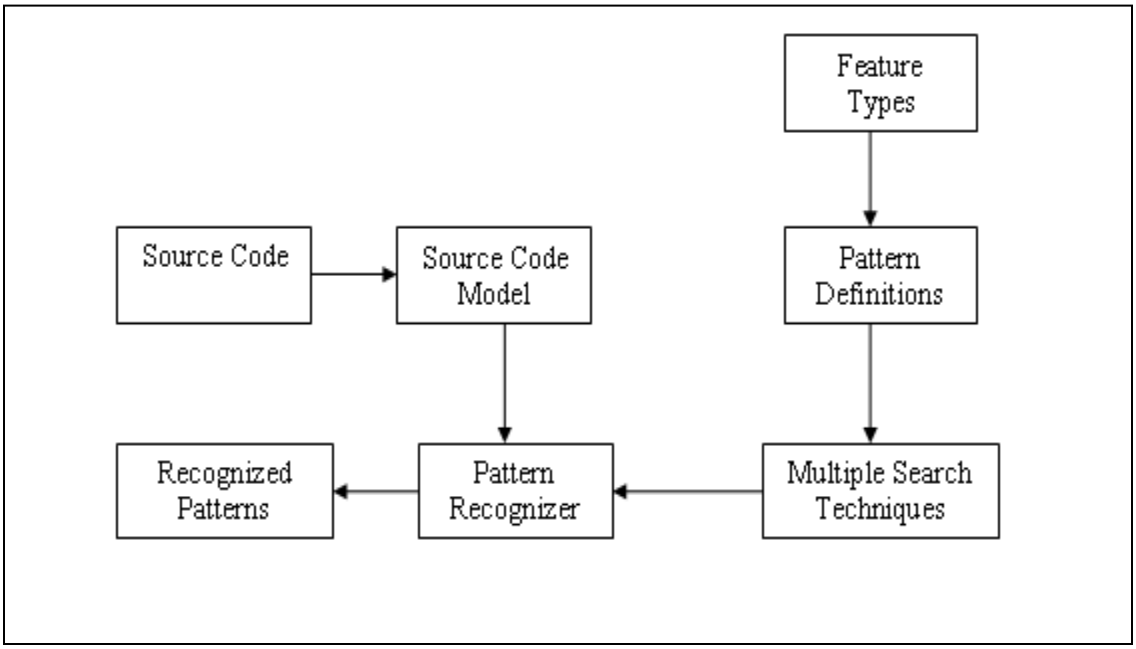


Figure 4.3: Overview of pattern recognition process

The source code is reverse engineered using Enterprise Architect Modeling Tool which creates source code model. Pattern recognizer uses multiple search techniques based on requirements of feature types and pattern definitions to recognize patterns. The detailed activities and techniques used for pattern recognition process are discussed in Chapter 6.

4.4 Challenges and Concepts of the Approach

The main concepts of the approach are briefly discussed in previous two sections. We come across different challenges and requirements for the implementation of different concepts introduced in Sections 4.2 and 4.3. This section discusses the challenges encountered and the basic concepts used to handle them.

Challenges

We came across the following challenges during different activities of pattern definition and pattern recognition processes.

Challenge 1: The key challenge for approach was to define and detect the variants of same pattern with accuracy. The order of feature types in pattern definitions play very important role for pattern definition and pattern recognition.

Challenge 2: The integration of multiple techniques in one integrated environment was another challenge for the development and implementation of prototyping tool.

Challenge 3: The extraction of static, dynamic, and semantic information is important to accurately recognize all types of GoF patterns. Initially, we extracted the static information related with design patterns using Enterprise Architect Modeling Tool [26]. The integration of static facts with dynamic and semantic information was challenging for pattern recognition process.

Challenge 4: The inheritance hierarchy used in the implementation of patterns was serious concern for our approach. It becomes difficult to write specific query that should extract accurate interface of any object, because user does not know the level of inheritance for the interfaces.

Challenge 5: The validation of extracted pattern instances was key challenge for measuring accuracy of our approach due to unavailability of trusted baseline results.

Concepts

We used the following concepts to handle the challenges listed above:

The concept of customizable and reusable feature types is used to define and recognize different structural and implementation variants of design patterns. This concept addresses challenge 1. The use of SQL queries and regular expressions allowed user to customize pattern definitions for variant detection. We do not claim 100% solution of this challenge, because there is still no agreed upon solution from community on common pattern definitions.

The key concept of the approach was to integrate the multiple techniques and use the power of these concepts in one integrated technique and tool. This concept is used to handle challenge 2. The concept of parameters in feature types is used to integrate the power of SQL, regular expressions, source code parsers and annotations. The application of annotations suggested in our technique nourishes the documentation of patterns and helps in pattern detection process.

The concept of alternative feature types provide flexibility to user for selecting alternative feature types for the recognition of same concepts implemented using different strategies. For example, the inheritance hierarchy challenge 4 is handled using this

concept. Alternative features types are also used for recognizing concepts of associations and aggregations in the source code.

We used the concept of SQL queries, source code parsers and regular expressions to extract static, dynamic and semantic information related with design patterns, which improve the quality of pattern recognition process. A number of patterns cannot be differentiated by using static and dynamic analysis techniques, because these patterns have similar structural and behavioral features. State and Strategy are examples of such patterns. Most pattern detection techniques use same definition and algorithm to detect such patterns. The semantic information is the solution to differentiate such patterns. The concept of naming conventions is used in [35] to handle semantic information. The uses of proper naming conventions depend on the discipline of programmers. We suggested/used the concept of using annotations to detect semantic information, which is helpful to improve accuracy and filter out false positives during pattern recovery process.

We selected strong, moderate and weak baselines approaches for the evaluation of our approach. The criterion for selection of baselines is given in Chapter 8. The parameters for selection of baseline's results address challenge 5.

The concept of multiple language pattern detection technique is also unique for our pattern recovery approach. The EA [26] support for reverse engineering source code of multiple languages and creating intermediate representation motivated us to create the multiple language parsers to extract the missing information directly from the source code. Design pattern recovery requires human intervention at some stages, because the intentions of designers for using any pattern to solve particular problem are not available in the source code. The patterns which are implemented based on intent hamper the recovery process of all pattern recovering techniques.

The approach suggests unique concept of annotations to add meta information related with each design patterns in the source code. By appropriate use of annotations, we cannot only detect the intension for using the pattern, but also improve the performance of approach by reducing the search space and time for the detection of patterns. The approach suggests/requests developers to use appropriate annotations for design patterns, which cannot help only in pattern detection process, but they are also helpful for maintaining documentation related with design patterns. The approach partially used annotations to show the improvement in its detection process. Due to lack of annotations in the source code, we use annotations as optional step. Furthermore, annotations need effort during forward engineering and maintenance cost is high, but annotations become effective over a period of time for recovery of patterns.

4.5 Summary

This chapter discussed the basic concept of approach which is base for the next two chapters. Pattern recognition approach is divided into two phases: pattern definition process and pattern recognition process. Section 4.1 discussed the abstract overview and major ideas used for the development of approach. The process used for creating pattern definitions is discussed in Section 4.2. Section 4.3 briefly discussed the pattern recognition process and the recognition technologies, which will be discussed in detail in next two chapters. The major challenges and concepts used to handle challenges are discussed in Section 4.4.

Chapter 5

Feature Types and Pattern Definitions

This chapter describes the feature types and the pattern definitions which are used for pattern recognition process. Feature types bridge gap between the pattern specifications and their implementation into programming languages. They play decisive role for creating pattern definitions which are used for the pattern recognition. The pattern definition process focuses on creating variant pattern definitions. The approach uses customizable and reusable feature types to create pattern definitions. The user can select predefined pattern definitions and is able to extend the catalogue of pattern definitions with new identified feature types.

Section 5.1 discusses feature types definitions, arguments of feature types and various varieties of feature types. A list of commonly used feature types is outlined. Section 5.2 discusses how to create pattern definitions using feature types. Finally, Section 5.3 explains the application of feature types used for creating pattern definitions by using Adapter pattern, Abstract factory method pattern and Observer pattern.

5.1 Feature Types

We described in previous chapter that approach used in this thesis is based on pattern definition process and pattern recognition process. The pattern definition process is based on feature types. It is important to discuss the structure of feature types and their arguments. Most patterns have been implemented using object oriented features like inheritance/realization, association, aggregation, delegation, etc. Developers use these object oriented concepts for the implementation of design patterns. We define patterns using programming language features which are comprehensible for experienced developers as well as for novice users. The main source for defining features of any pattern is to take its class diagram for static features and sequence diagram for dynamic features. All pattern specification methods use same source of information for design pattern specifications. We focus on defining feature types using different parameters which are generic and can be used for multiple languages. The feature types related with source code parsers are specific to one programming language. The used feature types and pattern definitions are customizable and flexible enough that user can add/modify feature types to handle the variants of patterns. A list of commonly used feature types is given in

Table 5.1.

Table 5.1: Feature types with brief description

FNo	Name	Description
F1	Has stereotype(I,C,M)	Means that any interface or class or method has specific stereotype.
F2	GetAllClasses(C)	Means extract all the classes.
F3	GetInterfaces	Means extract all the interfaces.
F4	Has Generalisation(C1,C2)	Means that class C1 extends class C2.
F5	Has Realization(C1,I)	Means that class C1 extends interface I.
F6	Has Association(C1,C2)	Means that class C1 holds reference of class C2.
F7	Has Aggregation(C1,C2)	Means that class C1 has aggregation relation with class C2.
F8	Has Composition(C1,C2)	Means that class C1 has composition relation with class C2.
F9	Has Delegation(C1,M1,C2,M2)	Means that method M1 in class C1 calls to method M2 of class C2.
F10	Has CommonSupertype(C1,C2)	Means that class C1 has same super type as C2.
F11	Has ChildClass(C1,C2)	Means that C1 is parent class and C2 is its child class
F12	Has CommonChildClasses(C1,Cn)	Means that C1 is parent class and it has common n child classes.
F13	Has Constructor(C1)	Means that class C1 has constructor.
F14	Has Common-Operation(C1,C2,M)	Means that classes C1&C2 has common operation M.
F15	Has Static Instance(A)	Means that attribute A is static.
F16	Has Pubic Operation(M)	Means that method M has public scope.
F17	Has private constructor(C1)	Means that constructor C1 has private scope.
F18	Has Return Type(M, T)	Means that method M has return type T.
F19	Has Return Value(M, V)	Means that method M has return value V.
F20	Has Parameter(M,Pn)	Means that method M has Parameters values P1,P2,P3……..Pn.
F21	Has Association label(C,I, L)	Means that any class or interface has association label L.
F22	Get ClassSource(C1)	Means extract the source code of class C1.
F23	Get MethodSource(M1)	Means extract the source code of method M1.
F24	Has Clone Operation(C,Co)	Means that class C has clone operation Co.
F25	Has FM Operation(C,FM)	Means that class C has factory method operation FM.
F26	Has TM Operation(C,TM)	Means that class C has template method operation TM.
F27	Has Annotation(I,C,M,S)	Means that any interface, class, method and statement has specific annotation.
F28	Match RegX(R)	Means match any regular expression.
F29	Has Type(V,T)	Means that variable V has type T.
F30	HasNotNullValue(V)	Means that variable V is assigned some value. It is not null.
F31	Has SameSignature(M1,M2)	Means that method M1 has same signature as M2.
F32	Has HeaderFile(C,h)	Means that class C has header file h.

There has been continuing debate over the language features that could make design patterns significantly easier to express [104 106]. The new features of programming languages such as (Java, C#, etc.) make the implementation of patterns easy by using (library methods, iterators, anonymous types, etc.), but these features complicate the pattern recovery processes. Judith Bishop [105] has sorted out complete set of implementations of patterns that make use of novel features of C#. The approach used in this thesis takes into consideration the new programming languages features for pattern definitions. Moreover, we also used helping feature types in our approach to supplement other feature types.

5.1.1 Arguments of Feature Types

Each feature type is used to add, remove and update features of patterns in order to control the implementation variants of different design patterns. Feature types have flexible and customizable structure based on static and dynamic parameters. The static parameters in feature types are used to reuse the same feature type across multiple searched patterns. The dynamic parameters relate a feature type with other feature types by taking results of one or more previous feature types. We used the following arguments in feature type definitions.

```
class FeatureType
  {
     string name;
     string query;
     int parameter;
     int CountOfPreviousResult;
    string searchMethod;
    bool reportResult;
  }
```

The “name” argument is used to give distinct name to each feature type. For example, has Stereotype (name) is the name of the feature describing that any particular class has specific stereotype. Stereotypes can store important information related with annotations which are helpful for pattern detection.

The “query” argument is used to write SQL query used to extract data for the feature type from the source code model. Each feature type is actually translated into SQL/REGX or parser module according to requirements. For example, “has Stereotype” feature uses the following query to extract all the classes which have stereotype Adapter.

Select Object_ID from T_Object where Stereotype=’Adapter’

The third argument "parameter" is used to check the status of parameters that will be used in each defined feature type. This parameter will refer whether the current query will use static/dynamic parameters. The queries take results from previous queries to match with features of patterns with the exception of only first query. For example, in the following feature type the value of argument "1" shows that it does not use result of any previous query.

("has stereotype", "select object_id from t_object where stereotype='%P0%'", 1, 0, "SQL", true);

"CountOfPreviousResult" parameter is used to check the number of results of previous queries used in the current query. Actually, we use different placeholders in the definition of different queries to include the results of previous queries. The number of parameters describe that how many results of previous queries are used in the current query. For example, the following query shows that PR0, PR1 and PR2 are previous results of different queries used in this query.

Select F.object_id, S.object_id, T.object_id, F.name from t_operation F, t_operation S, t_operation T where F.object_id=%PR0% and S.object_id=%PR1% and T.object_id=%PR2% and F.name=S.name and F.name=T.name and F.object_id<>S.object_id and F.object_id<>T.object_id.

"searchMethod" parameter plays significant role and is used to check whether we are using SQL query, regular expression or source code parser in feature type defination to extract features of patterns from the source code. Each feature type is translated into SQL query, regular expression or parser module feature according to the artifacts available in the source code model. We use regular expressions and parser module to extract artifacts directly from the source code. Firstly, the source code of any class or package is extracted from the source code model using SQL and then regular expression pattern definations are used to match the source code artifacts. The source code parsers are specially used to extract the information directly from the source code which is not available in the source code model. The regular expressions are fast to extract information from the source code direclty, but they fail to extract nested information from source code elements. So we developed and implementd the concept of parser module to extract such information which is beyond the scope of regular expressions. For example, EA Modeling Tool [26] is not able to extract delegation and method call information from the source code. We used Parser module/ regular expression patterns to extract such relations which are missed by Enterprise Architect Modeling Tool. The Visual Studio.Net Framework has full support for parsing regular expressions.

Finally, "reportResult" parameter is used to filter some feature types for the presentation of results. The feature types with false value of this parameter are used in the pattern recognition process to filter feature types for presentation of results. This parameter was required, because it is not important to display the results of helping feature types. The bool type is used to mention this argument. The true value of this parameter shows that current feature type is used in the pattern recognition process for the presention of results.

5.1.2 Negative Feature Types

We used the concept of negative features to filter out number of false positives during pattern detection process. When we add a new feature into any pattern defination, a boolean variable is used to check whether it is positive feature or negative feature. For example, in the case of Adapter design pattern, there cannot be inheritance relationship between Target and Adaptee class. In the same way, there cannot be inheritance relatioship between Adapter and Adaptee class. These kind of checks can improve the precision/recall of pattern detection process. Table 5.2 lists important negative feature types used in our approach. The complete list is given in Appendix C.

Table 5.2: Negative feature types

FNo	Name	Description
F1	Has No Generalization(C1,C2)	Means that class C1 has no generalization relation with class C2.
F2	Has No Association (C1,C2)	Means that class C1 has no association relation with class C2.
F3	Has No Aggregation (C1,C2)	Means that class C1 has no aggregation relation with class C2.
F4	Has No Realization (C1, I)	Means that class C1 has no interface I.
F5	Has No Constructor (C, M)	Means that class C1 has no constructor M.
F6	Has No Header file(C, h)	Means that class C has no header file h.

The Bridge pattern cannot have aggregation relationship between the refined abstraction class and concerete implementer class. The following feature type can filter out this relationship in the definition of Bridge design pattern.

Has No Aggregation(ConcreteImplementer, RefinedAbstraction)

The argument of negative feature is included in pattern feature definations instead of feature types, because it is optional argument. The syntax of pattern feature defination used in our approach is as:

```
Singleton.AddNewFeature(new FeatureOfPattern(FT7, new string[] { }, new int[] { 0 }, true));
```

Singleton is the name of pattern in which AddNewFeature function adds a new feature with name FT7 and type string. The new int[]{0} argument shows the sequence of

previous results of feature used within this new feature. The "0" value clarifies that FT7 is not using result of any previous feature type. Lastly, "true" indicates that it is negative feature type.

The application of negative feature types in our approach improve the precision and recall of pattern detection and reduce the search space for pattern detection.

5.1.3 Alternative Feature Types

The idea of alternative feature types is used to implement the concept of selecting more than one feature types from the catalogue of feature types according to implementation of same concept using different methods. For example, the interfaces of patterns are implemented using inheritance at different hierarchy levels. User does not know the exact level of inheritance implemented by the developers. We used the concept of alternative feature types to handle such problmes. We define different feature types for checking inheritance hierarchy within pattern definitions. Similarly, association, aggregation and composition have the same concept, but they are implemented in different ways which pose challenges for pattern recognition. The approach selects alternative feature types within single feature definition. Currently, we implementd the concepts of selecting alternative feature types using OR operation. For example, the interface for Adapter pattern may have following possible locations as shown in Figure 5.1. Adapter pattern definition uses alternative feature types using OR operation to specify all possible implementaions of interfaces.

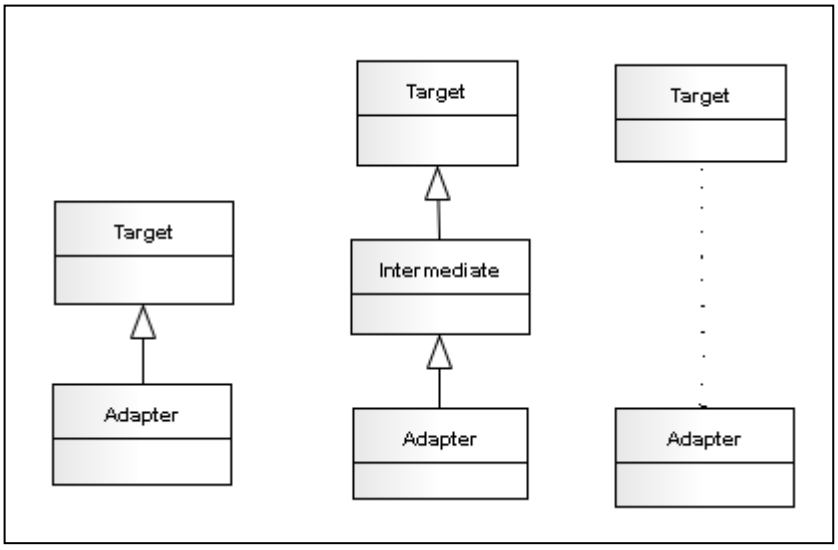


Figure 5.1: Alternative representations of adapter interfaces

5.2 Pattern Definition Process

Pattern definitions are created from selection of appropriate feature types which are used by the recognition process to detect pattern instances from the source code. Precision and recall of pattern recognition approach is dependent on the accuracy and the completeness of pattern definitions, which are used to recognize the variants of different design patterns. The approach follows the list of activites to create pattern definitions.

The definition process takes pattern structure or specification and identifies the major element playing key role in a pattern structure. A major element in each pattern is any class/interface that play central role in pattern structure and it is easy to access other elements through major element due to its connections. For example, in case of Adapter pattern, adapter class plays the role of major element. With identification of major element, the process defines feature in a pattern definition. The process iteratively identifies relevant feature types for each pattern definition. We illustrate the process of creating pattern definitions by activity diagram shown in Figure 5.3. The activity "define feature for pattern definition" further follows the criteria for defining feature type for pattern definition. It searches the feature type in the feature type list and if the desired feature is available in the list, it selects the feature type and specifies its parameters. If the catalogue do not have desired feature in the list, the process defines new feature types for the pattern definition. The process is iterated until the pattern definition is created which can match different variants of a design pattern. The definition of feature type checks the existence of a certain feature and returns the elements that play role in the searched feature.

The pattern definitions are composed from organized set of feature types by identifying central roles using structural elements. The pattern definition process reduces recognition queries starting definition with the object playing pivotal role in the pattern structure. The definition process filters the matching instances when any single feature type does not match desired role. The definition of Singlton used for pattern recogniton is given below in Figure 5.2.

```
PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] {
}, false));
        PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs1, new string[] { }, new
int[] { 0 }, false));
        PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new
int[] { 0 }, false));
        PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new
int[] { 0 }, false));
        PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs4, new string[] { }, new
int[] { 0 }, false));
        PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new
int[] { 0 }, false));
```

Figure 5.2: Singleton pattern definition

The above pattern definition is used for matching GoF structure of Singleton. The definiton can be customized for matching other variants of Singleton. We explain how Singleton pattern definition is created usign activity diagram shown in Figure 5.3. Firstly, we take Singleton specification given in Appendix B. The feature type FT1a is used to select all the classes participating in the examined model. We do not identify the major element for Singleton, because the GoF sturcure of singleton consists of only single class. The pattern definition process repeatedly checks that Singleton definition requires more features until the defnition is complete. The feature types (FTs1, FT1g, FTpc, FTs4 and FT7) are selected from existing catalog of feature types.

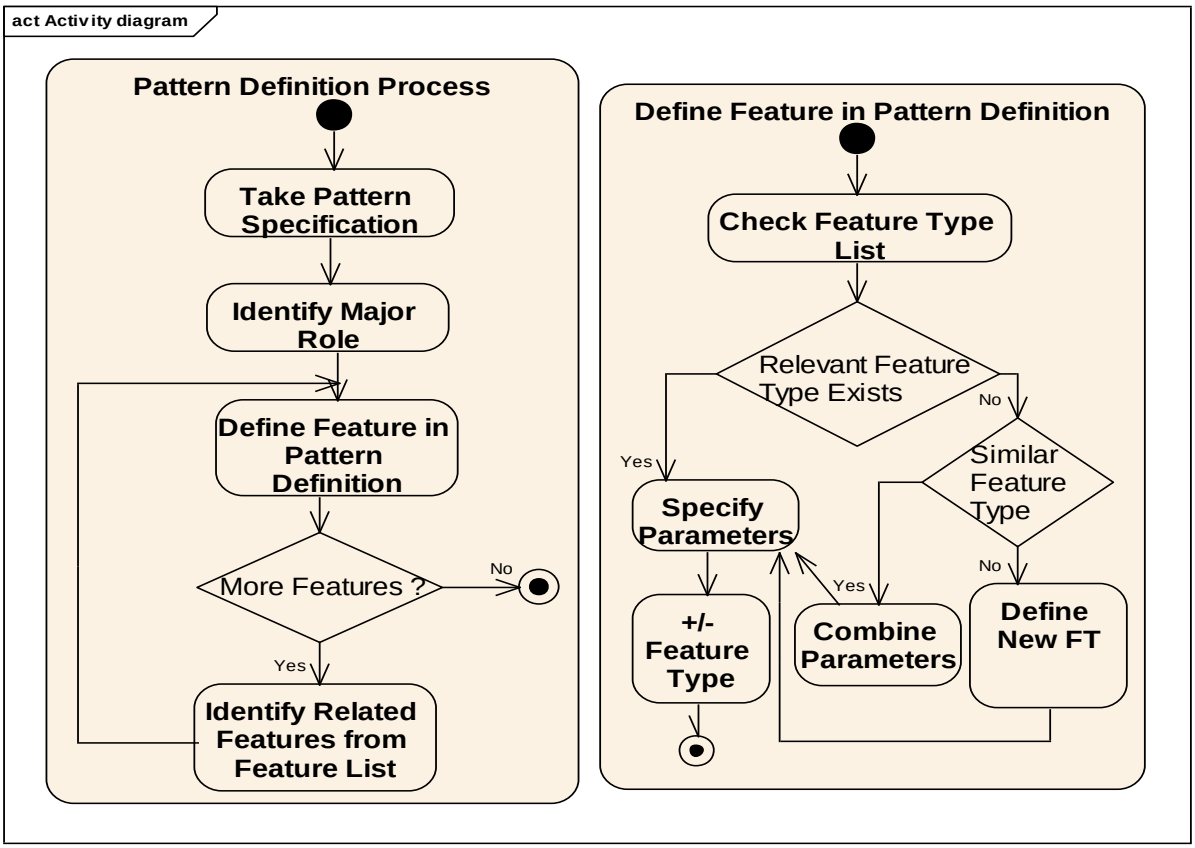


Figure 5.3: Activity diagram for creating pattern definitions

The pattern definition creation process is repeatable that user can select a single feature type in different pattern definitions. It is customizable in the sense that user can add/remove and modify pattern definitions, which are based on SQL queries, regular expressions, source code parsers to match structural and implementation variants of different patterns. The approach used more than 40 feature types to define all the GoF patterns with different alternatives. The catalogue of pattern definitions can be extended by adding new feature types to match patterns beyond the GoF definitions.

5.3 Examples of Pattern Definitions

We used pattern creation process to define static, dynamic and semantic features of patterns. It is clarified with examples that how features of a pattern are reused for other patterns. We selected one pattern from each category of creational, structural and behavioral patterns and complete list of all GoF pattern definitions is given in Appendix B. We describe features of Adapter, Abstract factory method and Observer in the following subsections.

5.3.1 Adapter

Adapter is used to provide compatible interface of one class into another interface according to the requirements of clients. It wraps the interfaces and provides abstraction for mapping different interfaces with each other. It can be used to transform data into various forms. Adapter, Proxy and Decorator have some common features, but Adapter provides different interfaces to clients, while proxy provides the same interface and Decorator provides enhanced interface. Adapter has two variations: Object Adapter and class Adapter as discussed below. The specifications for object and class Adapters are given in Tables 5.3 and 5.4. Another form of Adapter (introduced in Smalltalk, seen in JUnit Design) is pluggable Adapter. It is like object Adapter but it adapts to different adaptees with different interfaces. Pluggable Adapter uses reflection to access the adaptee method and invoke it indirectly in the Adapter. GoF structure of object Adapter is given in Figure 5.4.

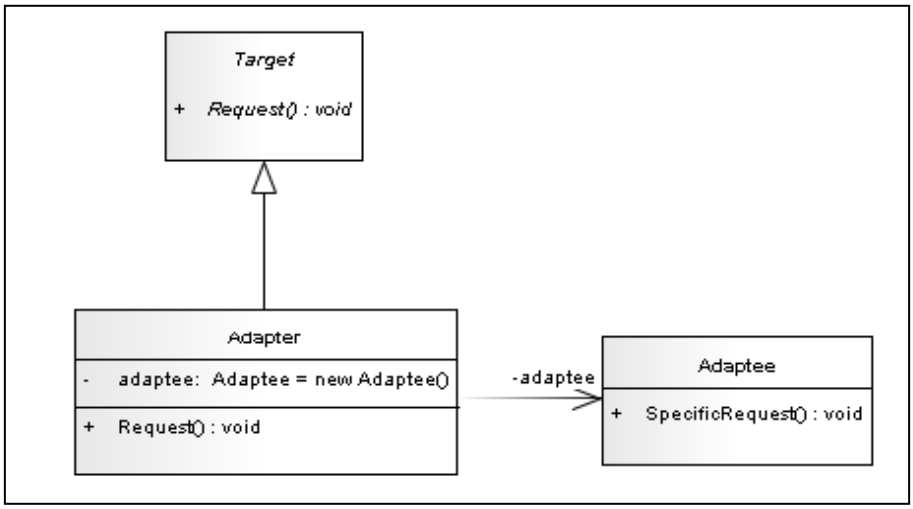


Figure 5.4: Adapter GoF structure [13]

Table 5.3 lists all feature types which are necessary to define characteristics of object Adapter pattern. The class Adapter variant can be defined by using the existing features of object Adapter and adding new features for variations.

Class Adapter can reuse (F1, F2, F4, F5, F6, F8) features of object Adapter and will add the following new features:

F10: Has Inheritance (C2, C3)

F11: Has Annotation (A1…….AN) $_{opt}$

Feature F3 and F7 are not required for the definition of class Adapter.

So the definition of class Adapter variant will have the following features:

Class Adapter (F1, F2, F4, F5, F6, F8, F10, F11)

Table 5.3: Object Adapter features

Features	Description
F1:Has Classes(C1, C2, C3)	It means that Adapter has classes C1, C2, and C3 with C1 as Target, C2 as Adapter and C3 as Adaptee.
F2:Has Inheritance (C2, C1)	It means that Adapter class inherits the C2 (Target class or interface).
F3:Has Association (C2, C3)	It means that Adapter class maintains the reference of C3 (Adaptee Class).
F4:Has No Inheritance (C2, C1)	It means that Adaptee do not inherit C1(Target).
F5:Has Delegation (C2, M1, C3, M1)	It means that Adapter class delegate request to real Adaptee class.
F6:Has Common Operation(C1, C2, M)	It means that Adapter and target have at least one common operation M.
F7:Has No Inheritance (C2, C3)	It means that Adapter and adaptee classes do not have inheritance.
F8: Has No Direct Access(C,C3)	It means that Client has no direct access to Adaptee.
F9:Has Annotations(A1…….AN)opt	It means check relevant annotations if available.

We clarify through example of class Adapter that how existing features are reused to create new pattern definitions. We further explain the reuse of existing features for Proxy pattern. The GoF structure of Proxy is described in Subsection 2.4.3 which has some similar features of Adapter. The name of parameter will be replaced with classes of Proxy pattern. For example, C1 = Subject, C2= Proxy and C=RealSubject.

Table 5.4: Class Adapter features

Features	Description
F1, F2, F4, F5, F6, F8	It reuse these features from object adapter.
F10:Has Inheritance (C2, C1)	It means that Adapter class inherits the C2 (Target class or interface).
F11:Has Annotations(A1…….AN)opt	It means that Adapter class maintains the reference of C3 (Adaptee Class).

The GoF structure of Proxy will share following common features of object and class Adapter patterns as given in Tables 5.3 and 5.4.

Common features: (F1, F2, F3, F5, F8, F10)

Proxy requires the following new features:

F12: Has Common Operation (C1, C2, C3, M)

F13: Has No Association (C2, C1)

Finally, the features of Proxy will be the following:

Proxy Pattern (F1, F2, F3, F5, F8, F10, F12, F13)

The other variants of Proxy as discussed in Subsection 2.4.3 can be defined in a similar method. The feature types used in above pattern definitions do not mention constraints on different artifacts playing roles in pattern definitions and pattern recognition. The constraints on pattern definitions are imposed during implementation of definitions in a programming language.

5.3.2 Abstract Factory Method

It provides an interface for creating families of related or dependent objects without specifying their concrete classes [123]. The Abstract factory method is beneficial because it isolates creation of objects from clients, but adding new product to factory needs changes in the interface of factory class and all of its subclasses. It can use Singleton, Factory method and Prototype for its implementation.

Table 5.5: Abstract factory method features

Features	Description
F1:Has Classes(C1, C2, C3, C4)	It means that Abstract factory has classes C1, C2, C3 and C4 with C1 as Abstract factory, C2 as Concrete factory, C3 as Concrete product and C4 as Abstract product.
F2:Has Inheritance (C2, C1)	It means that Concrete factory class inherits the C1 (abstract factory).
F3:Has Inheritance (C3, C4)	It means that Concrete product class inherits the C4 (Abstract product).
F4:Has Association (C1,C4, Client):	It means that Client class maintains the reference of C1 (Abstract factory Class) and C4(Abstract product Class).
F5:Has Delegation (C2, M1, C3, M1)	It means that Concrete factory class delegate request to Concrete product class.
F6:Has Common Operation(C2, C1, M)	It means that Abstract factory and Concrete factory have at least one common operation M.
F7:Has Created Products (M, P1…Pn)	It means that Method M of Class Concrete factory creates different products(P1…..Pn).
FT8: Has Return type(Mcp, C4)	It means that Method Mcp has return type C4 (Abstract product).
F9: Has Return value (Mcp,C3)	It means that Method Mcp returns C3(Concrete products).
F10:Has No Inheritance (C2, C1)	It means that Concrete product do not inherit C1 (Abstract factory).
F11:Has Annotations(A1…….AN)opt	It means check relevant annotations if available.

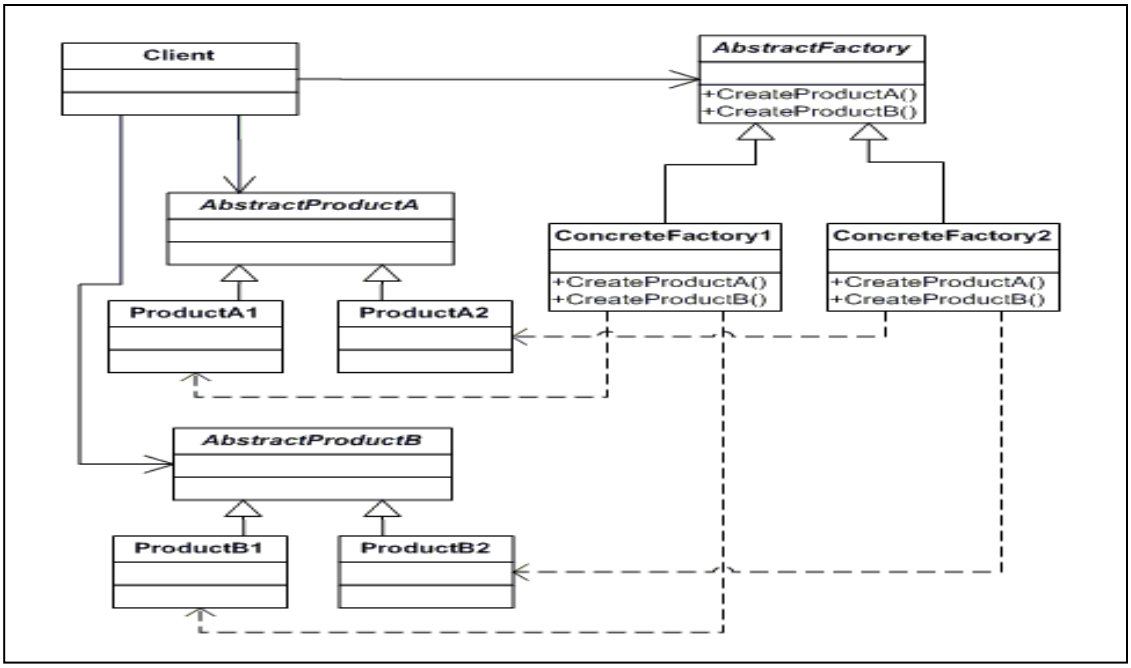


Figure 5.5: GOF structure of Abstract factory method [13]

5.3.3 Observer

Observer pattern defines one-to-many relationship between the subject and observer objects. When subject changes state, all the objects of observer are notified to update their state. It is used when subject do not know how many observers objects there are and it should be able to notify the objects without knowing these objects. It is implemented as part of Model View Controller as "View" and used in different GUI applications. It supports the event based applications involving multithreading. The GOF structure of Observer is given in Figure 5.6.

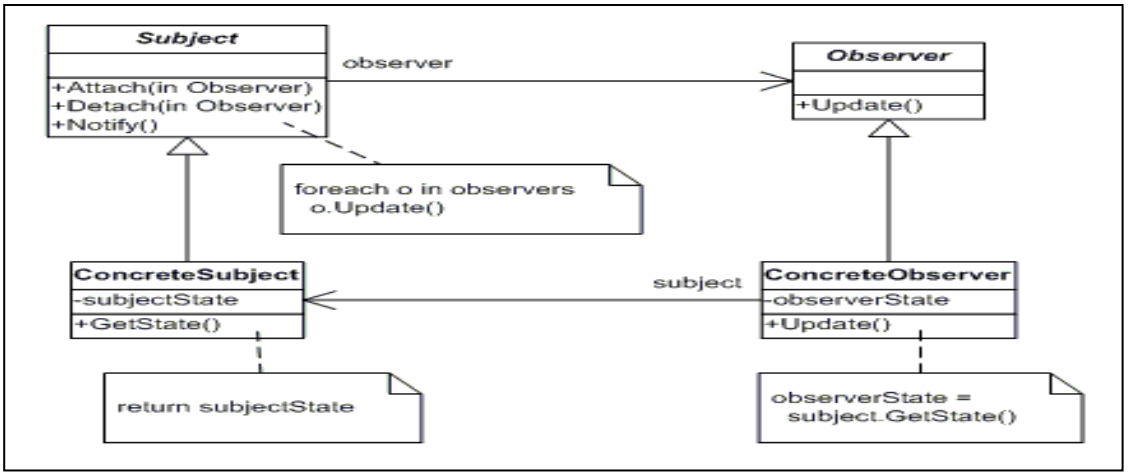


Figure 5.6: GOF structure of Observer [13]

Table 5.6: Observer pattern features

Features	Description
F1:Has Classes(C1, C2, C3, C4)	It means that Observer has classes C1, C2, C3 and C4 with C1 as Subject, C2 as Observer, C3 as ConcreteObserver and C4 as ConcreteSubject.
F2: Has Association (C1, C2)	It means that Subject class associates with C2 (Observer class).
F3: Has Association (C3, C4)	It means that C3 (ConcreteObserver) class associates with C2 (ConcreteSubject) class.
F4:Has Inheritance (C3, C2)	It means that C3 Inherits C2.
F5:Has Inheritance (C4, C1)	It means that C4 Inherits C1.
F6:Has Delegation (C1, M1, C2, M2)	It means that method M1 of class C1 calls method M2 of class C2.
F7:Has Delegation (C3, M1, C4, M2)	It means that method M1 of class C3 calls method M2 of class C4.
F8:Has Operations(C1,M1, M2,M3)	It means that C1 has methods M1 (Attach), M2 (Detach) and M3 (Notify).
F9:Has Operation(C2,M1)	It means that C2 has method M1 (Update method).
F10: Has Common Operation(C2,C3,M)	It means that C2 and C3 has common method M.
F11: Has return value(M, val)	It means that method M has return value (val).
F12:Has Annotations(A1…….AN)opt	It means check relevant annotations if available.

5.4 Summary

This chapter discussed feature types, feature type definitions, pattern definition creation and the concepts of different feature types (negatives, alternatives). We use these concepts for recognizing patterns in Chapter 6 and for implementation of our prototyping tool discussed in Chapter 7. Section 5.1 discussed the fundamental concept of feature types and role of feature types for pattern definitions. A subset of commonly used feature types is listed. Section 5.2 discussed the major concept of creating pattern definitions based on feature types. Finally, Section 5.3 discussed the examples of different pattern definitions. The concept of defining variant pattern definitions using predefined features from existing pattern definitions is elaborated. The next chapter elaborates the application of pattern definitions for recognition process.

Chapter 6

Pattern Recognition

Different techniques focus on fundamental definitions of design patterns during their pattern recovery process, but they lack support when a design pattern has been implemented using different style with same intent. With the increasing trend toward development of new applications using design patterns, the recovery of patterns from the legacy applications yield significant benefits in terms of quality, cost, time and human expertise for developing new applications. New patterns are developed which are applied in the specific areas such as software architectures, user interfaces, concurrency, security, services, etc. On the basis of complexity problems related with design pattern recovery techniques as discussed in Chapter 3, we focus on detecting different implementation variants of design patterns to improve the precision and recall of pattern detection process.

Section 6.1 describes objectives and scope of the approach. Section 6.2 discusses various search techniques used in the pattern detection process. The architecture and application of approach is discussed in Section 6.3. This section describes the static and dynamic view of approach. Finally, Section 6.4 discusses the strengths and limitations of the approach used for design pattern recognition.

6.1 Objectives and Scope of Approach

The overall objective of design pattern recovery approach used in thesis is to detect instances of design patterns accurately which can help the program comprehension, software maintenance, reverse engineering and reengineering of legacy applications. In order to achieve this goal, we used the following concepts.

6.1.1 Variants Detection

The accuracy of design pattern detection techniques is affected by not accurately detecting different variants of a same design pattern. See Section 2.4 for explanation of variants. The approach used in this thesis can recognize possible variants of design patterns due to flexibility of our pattern definitions. The semiformal definitions of design patterns with possible variants ease the novice user to define new pattern definitions and their variants. For example, the following variants of Factory method pattern are detected in JHotDraw 6.0b1 as mentioned in [111].

1 The factory method class extends creator and the factory method operation returns a new concrete product which extends the product.
2 The factory method class extends to its creator and the factory method operation returns a concrete creator to its creator.
3 The factory method class extends to its creator and the factory method operation returns the factory method class which extends its creator.
4 Parameterized factory methods may contain multiple product variants depending upon the selected parameters.

The variant definitions of Singleton and Proxy used for detection as shown in Table 6.1 are discussed in Section 2.4.

Table 6.1: Analysis of variants in different patterns

Pattern	Variant1	Variant2	Variant3	Variant4	Total
Singleton	1	2	0	0	3
Factory Method	81	6	3	0	90
Proxy	17	0	0	0	17

We managed the problem of variants detection by defining different variant pattern definitions as discussed in Chapter 5. The novel concept in approach is to integrate different techniques which can detect pattern variations. Users can customize pattern definitions in order to detect the structural and the implementation variants.

6.1.2 Algorithms for Pattern Detection

Different approaches use different algorithms for pattern recovery due to different definitions and various structural as well as implementation variants of a single design pattern. The numbers of approaches get intermediate representations from the source code which affect the algorithms for pattern recovery. We have analyzed the algorithms used in [3]. The algorithms are hard coded in the source code and they cannot be used to detect the variants of similar patterns. The sample algorithm for Proxy pattern used by approach [3] is based on fundamental structure and definition of Proxy as given in GoF [13]. This algorithm checks only that proxy class should have one interface and has association to one subject class. The algorithm fails when proxy class has number of interfaces and associations with number of realsubject classes. Similarly, the algorithms used in [25] for design pattern detection are hard coded in the source code. The approach fails to detect patterns when a pattern is implemented by using different variants.

We used SQL queries, regular expressions, source code parsers and annotations to match varying features of design patterns. The SQL queries and regular expression patterns are customizable and not hard coded in the source code of prototyping tool used for pattern recovery. The sample pseudocode for detection of Proxy pattern is presented in [111] as shown in Figure 6.1. Pattern detection algorithms use additional features of

delegation, aggregation and method return types detection supported by the source code parsers. The variants of different patterns have major focus during implementation of algorithms used for pattern matching. The feature types used in algorithms allow switching between different searching techniques for variants detection.

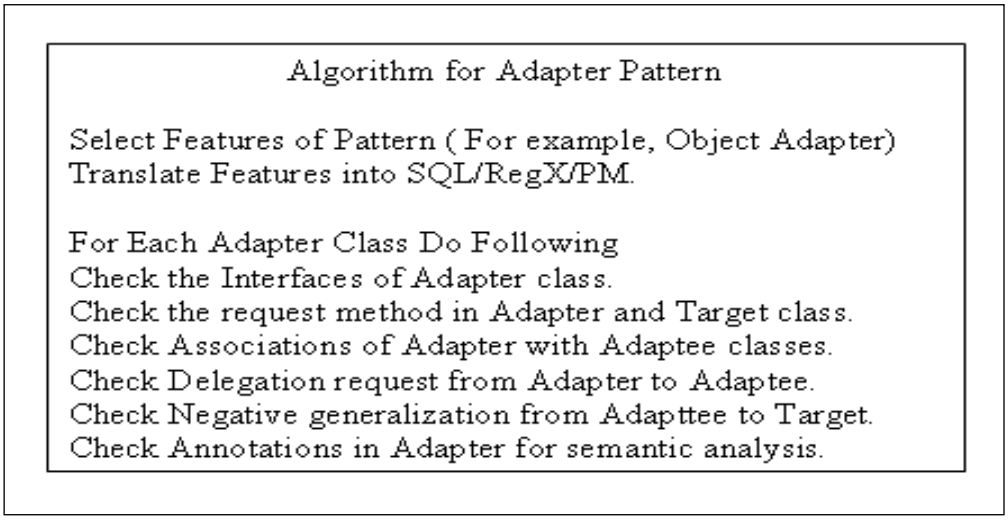


Figure 6.1: Algorithm for Adapter pattern detection

6.1.3 Overlapping in Design Patterns

Detection of overlapping is not trivial for reusability and comprehension of source code involving patterns, because some classes are used in different patterns and they play multiple roles. The overlapping in patterns prevents the increased artifacts playing roles in different patterns. We focued on detecting and visualiziang overlapped classes to support the maintainance and comprehension actitives. There exist different types of overlaps in design of different systems. For example, the creator class in the Factory method pattern and the subject class in the Proxy pattern can overlap. Similarly, during analysis of Apache Ant 1.6.2 [32], it has been observed that some classes are working as interface for composing different patterns. For example, the Task.java is used in different patterns (Adapter class in Adapter pattern, Component class in Composite pattern, Mediator and colleague class in Mediator, etc.).

Automated detection of overlapping with visual support is still overlooked area for design pattern research community which yields significant benefits to program maintenance and program comprehension. We detected overlapping in the source code of examined systems partially using only structural design patterns. The detected results are visualized using class model view, tabular view and report view. Figures 6.2, 6.3 and 6.4 show class view tabular view and report view of overlapping detected by the approach.

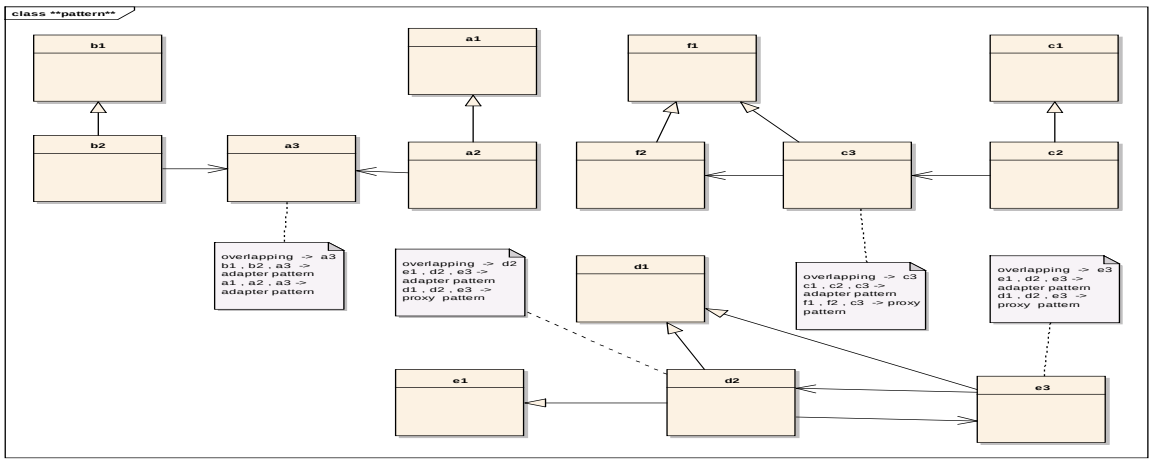


Figure 6.2: Overlapping in classes using class view

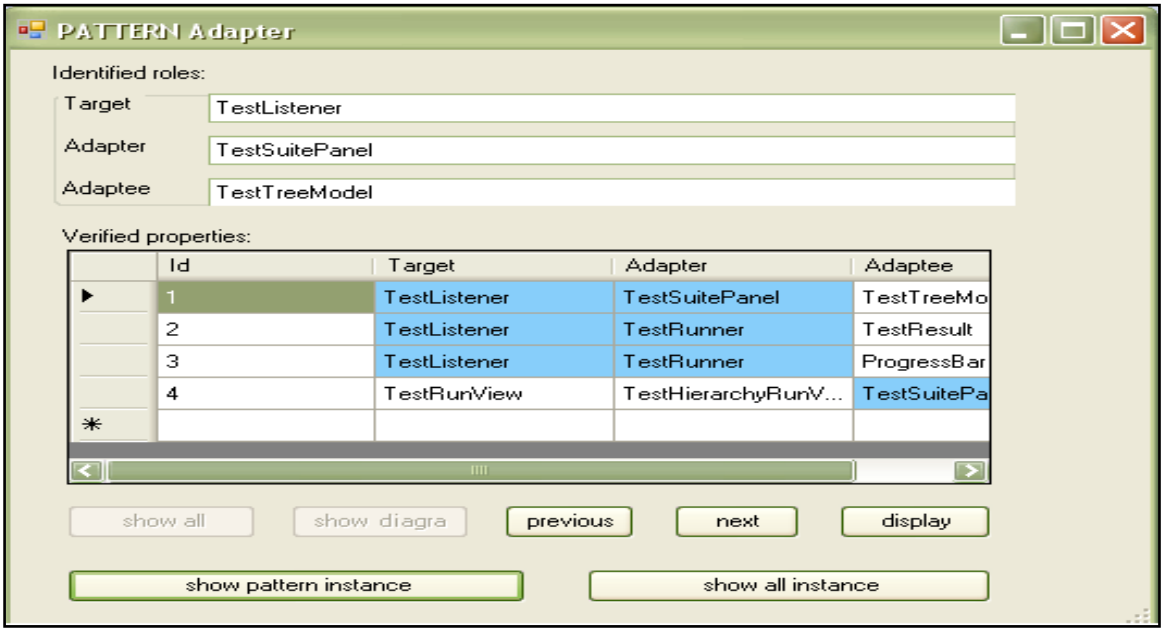
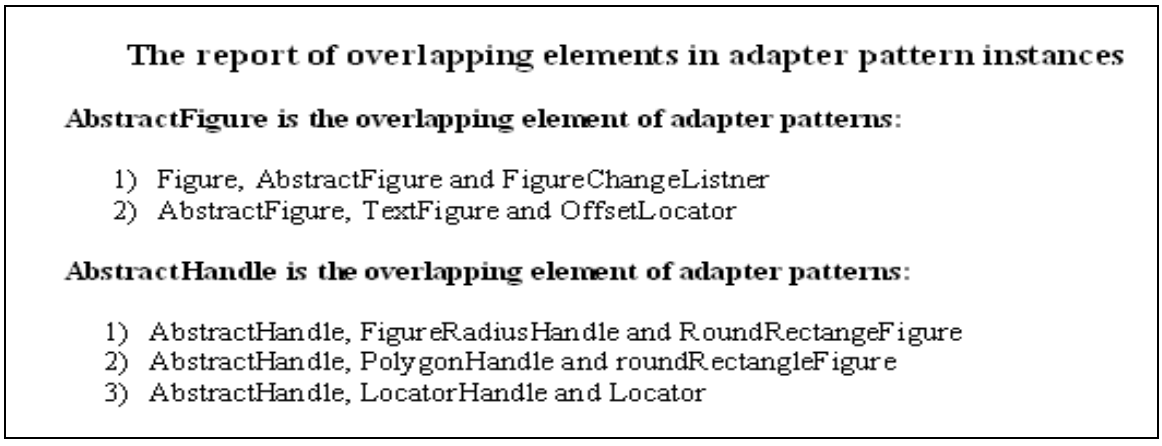


Figure 6.3: Tabular View of Overlapping Elements



The report of overlapping elements in adapter pattern instances

AbstractFigure is the overlapping element of adapter patterns:

1) Figure, AbstractFigure and FigureChangeListner
2) AbstractFigure, TextFigure and OffsetLocator

AbstractHandle is the overlapping element of adapter patterns:

1) AbstractHandle, FigureRadiusHandle and RoundRectangeFigure
2) AbstractHandle, PolygonHandle and roundRectangleFigure
3) AbstractHandle, LocatorHandle and Locator

Figure 6.4: Overlapping in Adapter pattern using report view

Lastly, Table 6.2 shows the overlapping of different classes in Factory method and Proxy patterns from JHotDraw 6.0b1. The ultimate goal of detecting overlapping is to make easier for the maintainers to know the impact of important classes particapting in different patterns.

Table 6.2: Overlapping analysis

Class	Factory method	Proxy
FileLocation: src/org/jhotdraw/samples/pert/PertFigure.java	PertFigure is a concrete Factory Method class	PertFigure is a proxy
FileLocation: src/org/jhotdraw/figures/AttributeFigure.java	AttributeFigure is a Factory Method class	AttributeFigure is a proxy interface
FileLocation: src/org/jhotdraw/samples/javadraw/JavaDrawApplet.java	JavaDrawApplet is a concrete Factory Method class.	JavaDrawApplet is a proxy
FileLocation: src/org/jhotdraw/standard/AbstractFigure.java	AbstractFigure is a Factory Method class	AbstractFigure is a proxy interface.
FileLocation: src/org/jhotdraw/figures/RectangleFigure.java	RectangleFigure is a concrete Factory Method class. RectangleFigure is a Factory Method class	RectangleFigure is a proxy
FileLocation: src/org/jhotdraw/util/collections/jdk11/ListWrapper.java	ListWrapper is a concrete Factory Method class	ListWrapper is a proxy

6.1.4 Composition in Patterns

Composition is important because different design patterns are used for plumbing the frameworks which are called architecture patterns. The architecture patterns are composed from different design patterns. For example, the Model-View-Controller (MVC) pattern [68] uses instances of the Observer pattern, the Strategy pattern, and the Composite pattern. The customizable pattern definition approach used in this thesis can define the architecture patterns and user defined patterns. We manually analyzed the composition of different patterns in the examined open source systems, but the automatic detection of composition is not in the scope of this thesis. The extracted composition among different patterns can be used to extract the design model information of legacy applications.

6.2 Multiple Techniques used for Pattern Detection

This section explains the role of various integrated techniques used to improve the precision and recall of presented pattern detection approach.

6.2.1 SQL Queries

Feature type definitions which are discussed in Section 5.2 used SQL queries as arguments to extract the relevant information required for extracting different patterns. Information about different artifacts of source code is scattered in different database tables created by the EA [26] Modeling Tool. See Chapter 7 for detail about EA. EA stores that information in more than 50 tables. This scattered information can be retrieved easily with the help of queries. For example, the following query is used to extract static information about the Singleton pattern.

Select t_object.Object_ID,t_object.Name, t_object.Scope,t_operation.Name,t_operation.scope,t_operation.Type,t_attribute.Name,t_attribute.Scope,t_attribute.Type,t_attribute.IsStatic from t_object, t_operation, t_attribute where t_object.stereotype='singleton' and t_object.Object_ID=t_operation.Object_ID and t_object.Object_ID = t_attribute.Object_ID and (t_object.name=t_operation.name or t_object.name=t_operation.Type) and t_attribute.type=t_object.Name and t_attribute.IsStatic=1 and t_attribute.scope='private'

The above query extracts the basic structural information for the Singleton pattern, but it cannot be used to extract all the variants of Singleton. Singleton is comparatively easy to detect as it has only one class with the exception of subclass Singleton, but its various implementation variants make its recovery difficult and challenging. The regular expression power for extraction of textual information enriches the SQL for retrieving the required information directly from source code. The multiple queries are required for matching properties of other patterns, because they contain multiple classes with different relationships. The execution order and dependence of queries on the other queries are very important concern for pattern recognition process. It depends on the requirement of current query that it want to use the result of one or more previous queries in specific order.

For example, the following SQL queries are used to match the features of Proxy pattern as mentioned in Section 5.3.

Query 1: "select object_id from t_object where stereotype='%P1%'"

Query 2: "select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (start_object_id=%PR0%" + ")"

Query 3: "select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (start_object_id=%PR0%) and (end_object_id=%PR1%)"

Query 4: "select end_object_id from t_connector where (connector_type='Association') and (start_object_id=%PR0%" + ")"

Query 5: "select F.object_id, S.object_id, T.object_id, F.name from t_operation F, t_operation S, t_operation T where F.object_id=%PR0% and S.object_id=%PR1% and T.object_id=%PR2% and F.name=S.name and F.name=T.name and F.object_id<>S.object_id and F.object_id<>T.object_id"

Delegation Parser call:"Has delegation in specific class", "%PR0%|%PR1%|%PR2%|%PR3%|%PR4%", 0, 5, "DelegationParser", true)

P1, PR0, PR1, PR2, PR3 and PR4 are placeholders which store the results of different queries. P1 by default stores the result of first query which is used to detect the stereotype of the artifacts used for pattern recovery. These results are used by other queries to match with different features of patterns. The delegation feature of Proxy is extracted by using source code delegation parser feature. The searching power of approach used in this thesis is unique, because each searching technique can integrate the results of other techniques. For exmaple, the following feature type integrates the results of source code parser and SQL query. SQL qeuery use results of source code parsers as arguments and further extract information from the source code model.

select object_id from t_object where object_type='Class' and name='%PR0%'

In above querie, PR0 argument is used to extract results from source code parser and SQL uses results of source code parser to extract information related with features of any pattern. Similarly, source code parsers use results of SQL queries as arguments.

The use of SQL queries for extracting static features of patterns from source code is fast, flexible and customizable mechanism, because user can add, delete and update queries instead of digging into the source code for modifications. The queries and feature types are independent from programmng languages, but queries are dependent on the information available in the source code model. To the best of our knowledge, there is no modeling tool that can extract all information from source code into a data model. Due to limitation of modeling tools, we used the concepts of regular expressions and parser module to extract missing information which is important for pattern recovery.

6.2.2 Regular Expressions

Regular expressions are a notation which is suitable for describing syntactic tokens in a language [107]. They have been used in programming and different text editors from long time for matching text. They have context independent syntax and can extract line by line information by scanning the source code. We used regular expression pattern matching in our technique for extracting artifacts which are not available in the database model created

by the EA [26]. The simple syntax and flexibility of regular expressions allow user to extend the pattern specifications according to his/her requirements. Regular expressions have a clean and declarative semantics that provide a mechanism to select specific strings from a set of character strings. The various operators and meta characters are used as standard across most implementations of regular expressions. Figure 6.5 lists meta characters, class short hands and POSIX class definitions mostly used by regular expressions. For example, the meta character "+" denotes one or more occurrences of any character string and "\s" class short hand is used to match spaces in different strings. Similarly, the POSIX class definitions are used to denote common ranges. For example, the POSIX class definition [: digit:] will match digits from 0……..9.

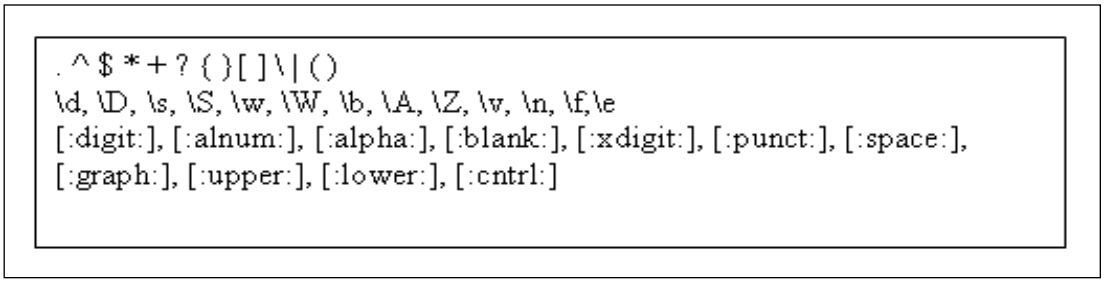


Figure 6.5: Meta characters and class short hands

We used abstract regular expression definitions which allow the extension of vocabulary of regular expressions by using the definition of one regular expression in another pattern. In this way, we extended the vocabulary of our pattern definitions. The abstract features of regular expression syntax make easy for the user to refine different pattern specifications according to the nature of source code written in different languages. For example, the following regular expression uses abstract pattern definitions to match java method declarations from the source code.

JMethodAccessSpecifier)?\s*(JMethodModifiers)?\s*((Types)\s*(\w)+\s*\(\s*(.*)\)\s*(throws)\s*(\w+)\s*)

In the above pattern specification, the definitions of JMethodAccessSpecifier, JMethodModifiers and Types are abstracted. The actual pattern specifications of these abstracted constructs are given below.

JMethodAccessSpecifier: friendly| public|protected|Private|private protected

JMethodModifiers: synchronized|native|final|abstract|static

Types: char|int|float|double|long|short|boolean|byte|void|string

The level of abstraction depends on the requirement of user. For example, in the above pattern specification the definition of synchronized in the JMethodModifier is abstracted.
We have developed a custom build tool DRT [108] which uses regular expression definitions to match the different artifacts available in the source code. The regular

expression pattern definitions are independent from implementation and can be used in any text editor and parser which have regular expression parsing features. The technique used DRT [108] for pattern matching as shown in Figure 6.6. The artifacts extracted from source code of different programming languages are shown in Table 6.3. For the purpose of extracting design pattern features like delegation, lazy initialization, method invocations, we used the vocabulary of our regular expression pattern definitions in the Visual Studio.Net framework. The support of regular expression parser makes it possible to use the pattern definitions of DRT[108] in our pattern recognition prototype.

Finally, the regular expression pattern specifications can use the results of SQL as paramaters in their syntax. This mechanism allowed us to utilize the full power of regular expressions with SQL queries for retrieval of desired artifacts from the source code. For example, the following regular expression pattern calls SQL in its defination to check method call in the body of another method.

public|protected)?\\s*\\w+\\s*%PR0%\\(.*\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*\\w+\\.%PR1%\\(\\)\\;

PR0 is the name of calling method and PR1 is the name of called method in the above pattern specification. The results of these arguments are given as input to pattern defintions by using queries. In the same way, the SQL queries can use the results of regular expression features for extracting pattern related information.

We explain extraction of artifacts from source source using regular expressions through Figure 6.6. The engineer analyses source code and designs the pattern specification describing information to extract the desired artifacts. The engineer may refine the specifications and reapply to extract new artifacts. The abstract pattern specifications are used to extend the vocabulary of different pattern specifications, which is the major challenge for lexical pattern matching techniques.

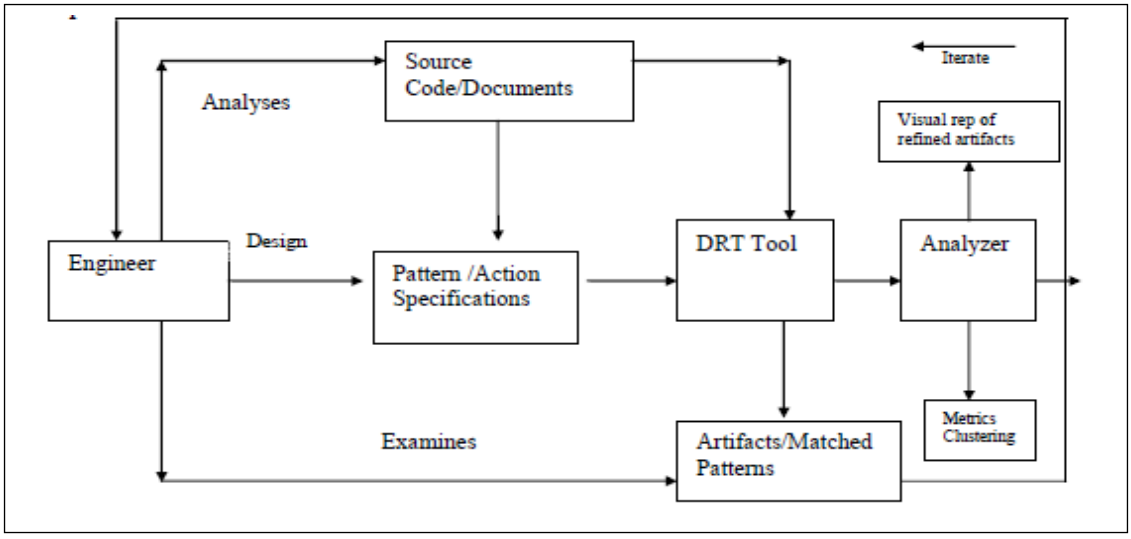


Figure 6.6: Overview of regular expression pattern extraction technique [109]

The approach is able to extract artifacts from the source code of different languages. At primary level, these artifacts provide statistics about the structure of legacy applications which can be used further for detail analysis. Some tools are not able to process the source code, which have syntax errors or miss some header files or have mix-code, but DRT can extract artifacts from the source code with such constraints. The pattern definitions of DRT have support of action and analysis patterns to filter out wrong matches during execution of any pattern specification. The process of action, analysis and filtering features is explained in [109].

Table 6.3: Artifacts extracted using regular expression [109]

Software	Size on disk	LOSC KLOC	Files		Include files	Functi-ons	Blank lines	Lines of Comments
			Total Files	Code files				
Alligance Game/C++	823MB	450	7629	1341	3463	612	71964	74679
Elm/C++	8.05MB	35	479	455	905	422	6566	7686
Tac_Plus/C	592KB	20	50	50	153	310	3181	2600
Mining/Java	150KB	5	5	5	11	121	684	1088
Monica/VB	2.50MB	18	50	33	-	621	5	50
Drawing	1.53MB	8	45	10	-	252	847	524

6.2.3 Source Code Parsers

The motivation for developing source code parser module came due to limitation of regular expressions for matching nested information from the source code and missing capability of reverse engineering tools used to create the intermediate representations. The parser module is divided into the following sub-modules:

I) Delegation Parser
II) Aggregation Parser
III) Method Invocation Parser
IV) Return Type Parser
V) Method Invocation through Reference Parser etc.

The architecture of parser module is shown in Figure 6.7. The grammar file is given as input to Coco/R parser generator as input. The grammar for each language is available on web. The Coco/R parser accepts the grammar for each language and creates the parser and the scanner for that language. The robust scan parser generator investigates the source code for a particular language and creates different type of parsers. Currently, parser module supports C# and Java languages, but it can be extended for the other languages. Furthermore, the scanner and parser for each language are used to create different

language parsers which are used as feature types for detection of patterns. The purpose of each parser module feature is explained below:

Delegation is the most important property which is used to implement number of design patterns. Modeling tools are not able to detect the delegation information from the source code. The implementation of delegation by different ways in different open source systems makes its recovery challenging. In order to handle the challenge of recovering delegation from source code artifacts, we developed and implemented delegation parser module. For example, the delegation parser module for Adapter pattern takes Adapter class name, name of request method and source code of adaptee from the SQL queries and it detect the delegation call in the request method of Adapter class. The parser is capable to detect the call with the line number in the source code. This detailed information is used for rapid verification of design pattern properties and comprehension of the source code. The delegation parser module is unique, because it can extract the delegation call from source code of Java, C# and C/C++ languages. The steps required for implementation of delegation parser are further explained by an algorithm in Figure 6.7. To the best of our knowledge, the delegation property is only language specific and most of approaches are not able to recover delegation from multiple languages.

Aggregation is the important characteristic which is used to implement number of design patterns. Unfortunately, UML tools are still not able to differentiate between association, composition and aggregation. The tools differentiate these concepts for forward engineering, but they are not able to detect how these relations are implemented in the source code. The purpose of aggregation parser module is to support our prototyping tool to detect aggregation relation from source code of different languages. Aggregation parser takes the name of class as argument and checks the aggregation of all the classes with the aggregating class. The output of aggregation parser is used in SQL queries for extracting features of Decorator pattern, Bridge pattern Composite pattern, etc.

Method Invocation Parser module is used to detect the overriding methods which are called by other methods. For example, Template method pattern uses the primitive operations which are called by the template method operation to perform the required function. The parser takes the name of class, calling method and called method as arguments and return true/false based on the invocation of call. The method invocation parser module supports the Template method pattern. Similarly, method invocation through reference parser is used to check the invocation of method of one class within another class through reference. It takes same arguments as method invocation parser with the addition of second class name. This source code parser is used for recognizing Visitor, Observer, etc.

Finally, Return Type Parser is used to detect the return type of methods and their returning values. Some methods create different objects and return these objects. Singleton, Factory method, Builder and Abstract factory method patterns use features of

this module. The parser takes the name of class and method as arguments and checks the signature of methods. The method data types and return values are compared and it returns true if both are equal. In the other hand, it checks the instance of classes within body of methods and returns this instance.

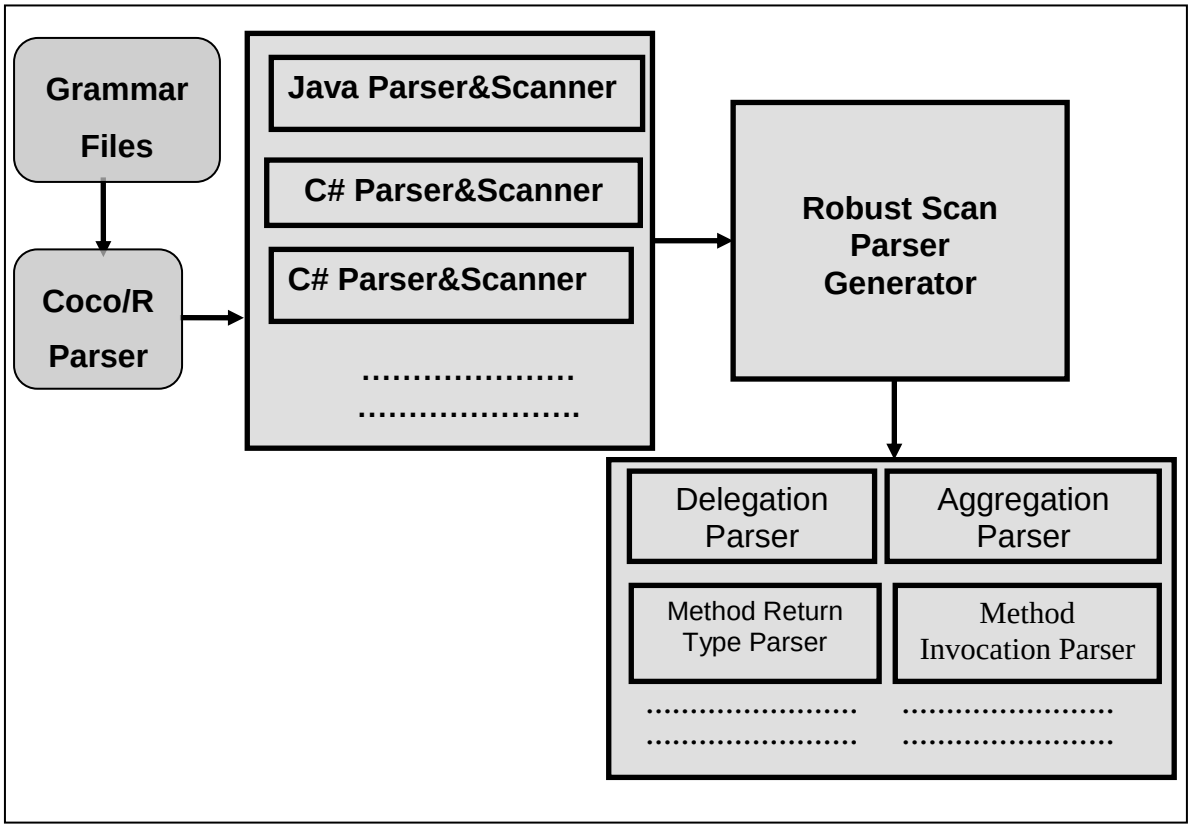


Figure 6.7: Architecture of parser module

The approah used specialized source code parsers which extract required information from specific source code artifacts. Each parser can use results of other parsers for extracting required properties of design patterns available in the source code. We extracted the dynamic properies of design patterns by using source code parsers. The output of parsers are also used within SQL queries and regular expressions as parameters to extract required information. The parsers are implemented in the tool as separate feature types.

Delegation Parser (DelegatingClass, RequestMethod, DelegatedClass)

1 Input source code to Coco/R generated Java source code parser.
2 Retrieve token stream from Coco/R parser.
3 Read the token stream to search a token (name of DelegatedClass)
4 Check if found token is relevant for class definition.
5 Search beginning and ending token of the DelegatedClass.
6 Extract non-private method definitions with these tokens.
7 Store found method definitions in a list.
8 Read the token stream to search a token(name of DelegatingClass)
9 Check if found token is relevant for class definition.
10 Search beginning and ending token of the DelegatingClass
11 Search DelegatedClass name within these tokens.
12 For each found token check that they are required to create an instance of DelegatedClass and store the found reference tokens in a list (Li1)
13 For each found token check if it is surrounded by brackets (checking for typecasting) and store the found token in a second list (Li2)
14 Search the name of RequestMethod within beginning and ending token of the DelegatingClass.
15 For each of the found tokens in List Li1:
   a) Check if token is relevant for method definition.
   b) Find beginning and ending token of the method.
   c) Search Within beginning and ending token of the method a reference of a token stored in List Li1.
   d) For the found token, check a call to method of DelegatedClass through appropriate reference.
16 For each of the found tokens in List Li2:
   a) Move to the token in the token stream that is type casted
   b) For this token, check if the value of the token that is two tokens behind the found token equals the name of a method within the DelegatedClass.

Figure 6.8: Algorithm for detecting delegation in Java source code

6.2.4 Annotations

The SQL queries and regular expressions are fueled by the use of annotations, which play role in our approach in the context of forward engineering for the purpose of documenting the patterns and in reverse engineering for the recognition of patterns. Annotations add metadata information to different artifacts in the source code and this information can be processed by different tools (compilers, javadoc etc.). The existing mechanisms of using annotations have been adequate for general purpose such as documentation and refactoring of source code, but they do not support in complicated uses and particular have a lack of support for design pattern recovery. The programmers use annotations for description of patterns which are not sufficient for pattern recovery. Annotations contain intensions of developers which can be used to document the patterns and provide support for the recovery of design patterns from the source code.

The annotations defined in [110] are used for selection of particular block of the source code with special focus on refactoring the source code. We have defined annotations which are used for pattern documentation and recovery. The defined annotations could be used by machine as well as by human. The machine configurable part can be used by the compiler for detecting errors or suppressing warnings. The human part is used for JavaDoc to maintain the documentation of the artifacts and changes made in the source code. We have defined more than 50 annotations which reflect the intentions of developers in the source code. We are extending and modifying the list of annotations defined in [110] to detect the similarity of different annotations used in multiple patterns. Due to limitation of space, the following subset of annotations is shown as sample for maintaining documentation and recovery of design patterns. The detailed list of annotations is given in Appendix E.

1) @abstract {notification | [interface, access_to_subsystem] | state_management | list_traversal | object_identity | handling}

2) @compose {object} from {different_objects | related_objects}

3) @decouple {receiver} from {sender}

4) @object {instances} {share_by_introducing} {state_handlers | intrinsic_state | central_instance}

5) @decouple {implementation} for{dynamic variation lists}

6) @decouple {sender} from {receiver}

7) @provide {handlers} for {requests | expressions}

8) @ instantiation {eager|lazy|replaceable}

9) @ interface {adapt | enhance | simplify}

10) @flexibility for {object_creation | guts}

11) @dynamic_handling of {object_creation | requests | expressions | configuration | implementations}

12) @flexibility for {object_creation | guts}

13) @provide {handlers} for {requests | expressions}

14) object {change} {skin | guts}

15) @traverse {object_list | composite_list}

16) @object {creation} {flexibility | [clone, copy] | [build, compose] | speed-up | control | remotely | restrict_instance_count | simplify}

Table 6.4 lists subset of annotations that relate to specific design pattern. The developer can use these annotations in the source code as guidelines for the documentation and the maintenance of legacy systems.

Table 6.4: Group of annotations related to design patterns

Pattern/Annotation	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Singleton		x	x	x				x								
Composite		x		x		x				x		x				
Adapter	x								x							x
Bridge									x						x	
Factory Method			x		x			x							x	
Proxy	x		x				x		x		x		x	x		
Decorator	x					x			x							
Facade										x	x	x				

The concept of human and machinge readable part of annotations was used in [110] for refactoring the source code. We extended the concept used by the technique to support detection of design patterns. The combination of human as well as machine readable part for annotations is important for our pattern recovery approach. Our pattern detection process matches the similarity between the identification number for the annotation in the source code and in annotation.type file. Our intention is to use the human readable part of annotations for the static analysis of source code and detection of structural design patterns. The machine configurable part will be used for dynamic analysis. The combination of human as well as machine semantics of annotations is very helpful for our pattern detection process. Figure 6.9 shows the application of MID (Machine Identification number of annotation and HID (Human Identification number) for detecting Adapter and Factory method operations in the source code.

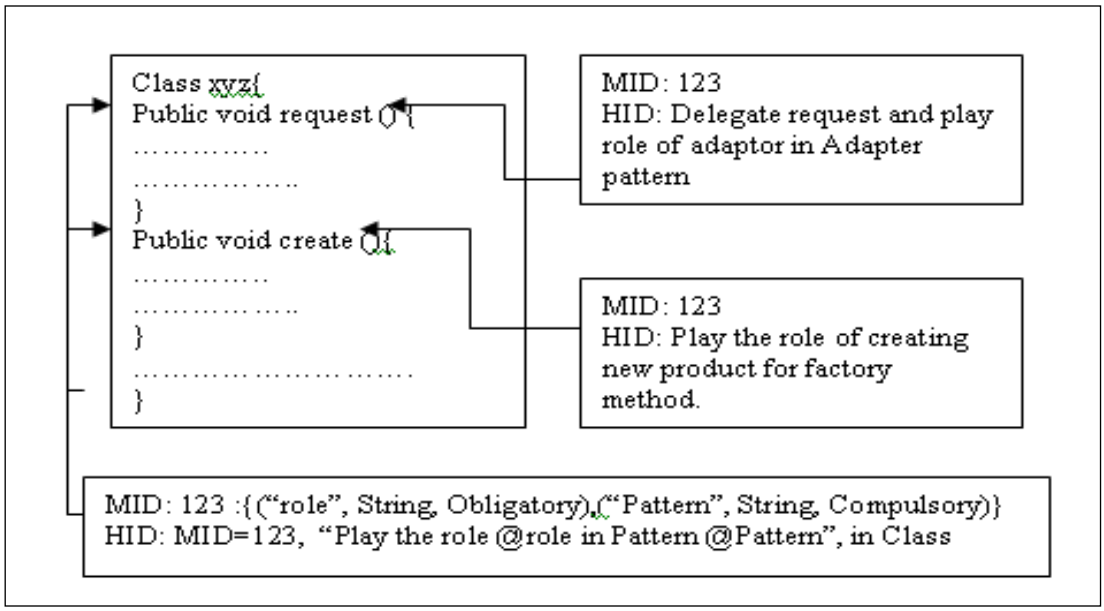


Figure 6.9: Annotated class in source code with annotation. Type [111]

6.3 Architecture and Application of Pattern Recognition Approach

The pattern recovery approach used in this thesis is based on integration of Structured Query Language (SQL), regular expressions, source code parser module and annotations. Feature types use multiple searching techniques as parameters to extract patterns using source code model, which contains database model as well as UML model of the source code. The database model contains important information about the structure of various artifacts used in the source code. For example, it contains structural information about important artifacts in source code such as classes, interfaces, methods, attributes and relationships between these artifacts. The important relationships extracted by EA [26] are generalizations, associations, realizations, method parameters, method scopes, method return types, attribute types, etc. This information is very useful for realizing basic properties of design patterns, but it is not worthwhile until additional information is not available to detect creational, structural and behavioral design patterns.

The UML model represents relationships between the artifacts in form of class diagrams. It is questionable that why we selected EA for intermediate representation because large number of other tools are available? Firstly, we used EA tool for the intermediate representation of the source code due to its excellent capabilities for reverse engineering source code and plug-In support with .Net Framework. Secondly, we have also prior experience of using tool for different projects. Thirdly, EA is capable to reverse engineer the source code of more than ten languages. Finally, the standard output formats of EA can be used for the visualization of our results.

The SQL queries are used to extract the information available in the database model and then this information is further used to match properties of different design patterns. Delegations, aggregations and method invocations are very important features of design patterns as mentioned in Subsection 6.2.3. These relations are missed by EA Modeling Tool during reverse engineering of source code. We used regular expressions/ source code parser module for extracting such relationships from the legacy applications which are missed by the EA tool. The detailed application of regular expressions used in approach is discussed in Subsection 6.2.2. The source code parser module is developed to extract the information which is beyond the scope of regular expressions. Delegation is the key feature of patterns and delegation parser is used to extract the delegation information from the source code. Delegation parser module reports the delegation request from one object to another object with the line number in the source code where actual call is invoked. The other functions of source code parser module are discussed in Subsection 6.2.3. Finally, annotations play very important role in our approach, because they are used to describe the semantics of the code which are important for the documentation of patterns as well as for the recovery of design patterns. The detail about the application of annotations is

discussed in Subsection 6.2.4. The framework of our pattern detection approach is shown in Figure 6.10 which follows the following steps:

I) The approach takes source code and uses EA for creating intermediate representation of the source code.

II) Pattern definitions based on reusable feature types are selected for the recognition of desired pattern using pattern definition creation process.

III) The recognition technology is selected according to arguments in feature types used for pattern recognition.

IV) Finally, approach presents the recovered results.

The dynamic view of recognition process with major activities is illustrated in Figure 6.11 which follows the above mentioned steps. We explain the recognition process by an example shown in Figure 6.12. Suppose Enterprise Architect Modeling Tool has created the class model of source code. The elements in the model have various relationships and current sketch do not match with any GoF pattern structure.

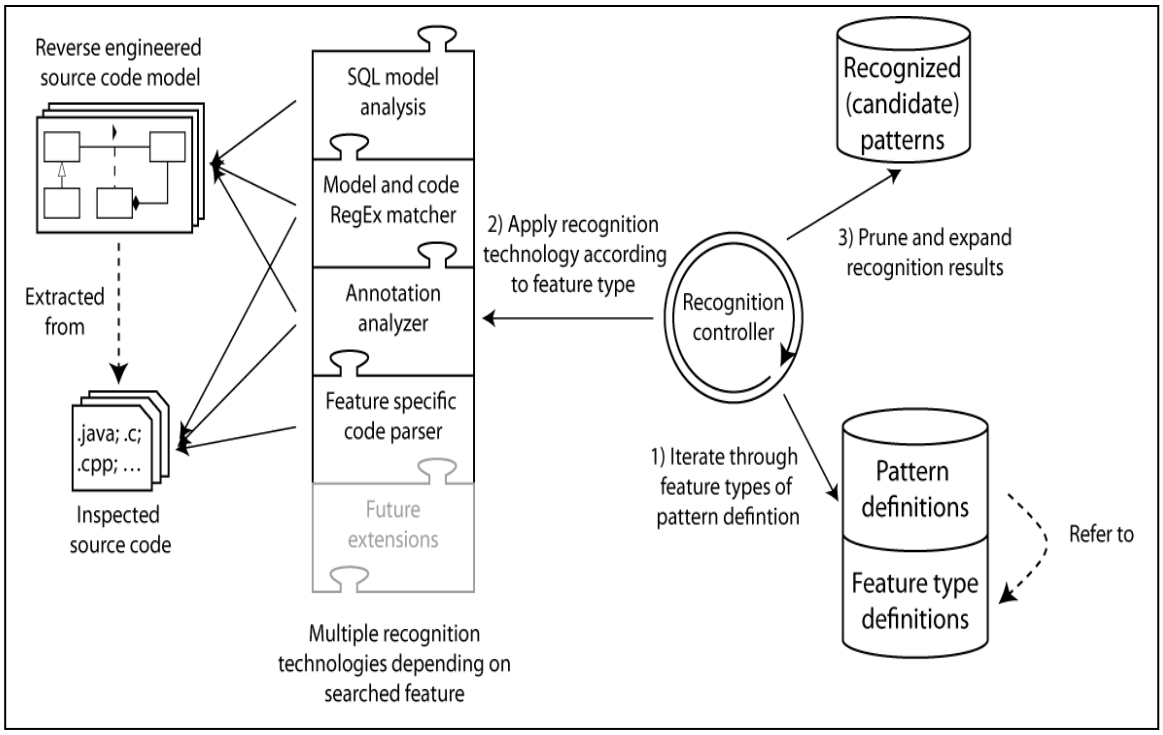


Figure 6.10: Static view of pattern recognition process

We apply factory method pattern definition on this structure. Our approach first selects all the elements participating in this model with the help of appropriate feature type. Initial feature type uses SQL query with appropriate parameters to extract all elements.

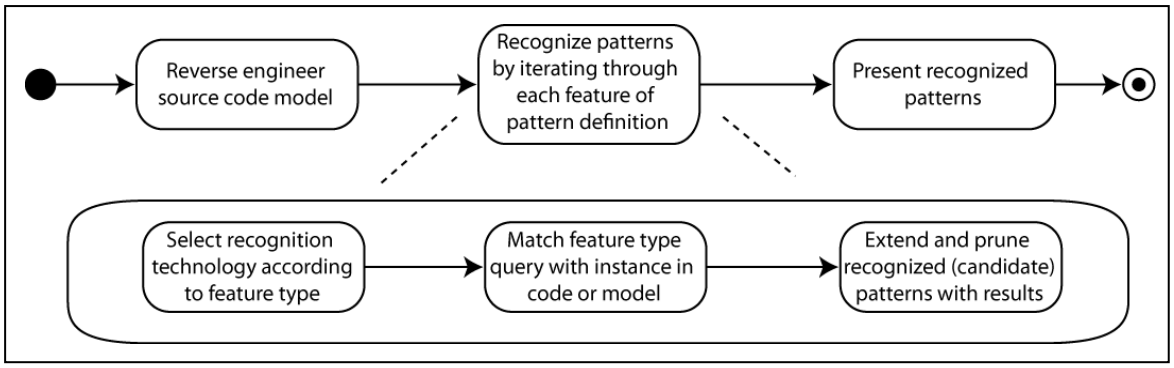


Figure 6.11: Dynamic view of pattern recognition process

Secondly, the approach matches elements which have generalization relation with the other elements. The feature type (Has Generalization) selects elements F, E, C, B, G and filters all other elements. Thirdly, we look for common operations in the elements which have generalization relationships. The common operation feature type searched at least one common operation in pairs (F,E) and (C,B). As a result of this feature type, the elements F, E, C, B stay in the model and G is filtered out. In the continuation, the process checks that common operation in element F creates an object of another class and returns it. According to model, this role is played by operation of F element. The source code parser module feature is used to perform this operation. Finally, the recognized model maps with the features of Factory method pattern as discussed in the specification Subsection 2.4.2. The order of feature types in the pattern definition play important role to improve the performance of approach. It is questionable that which element should be selected as staring element for recognition of pattern. We select starting element for each pattern according to roles of individual elements in each pattern. For example, in case of Adapter pattern, we select Adapter class as staring element due to its major role and connections with other elements.

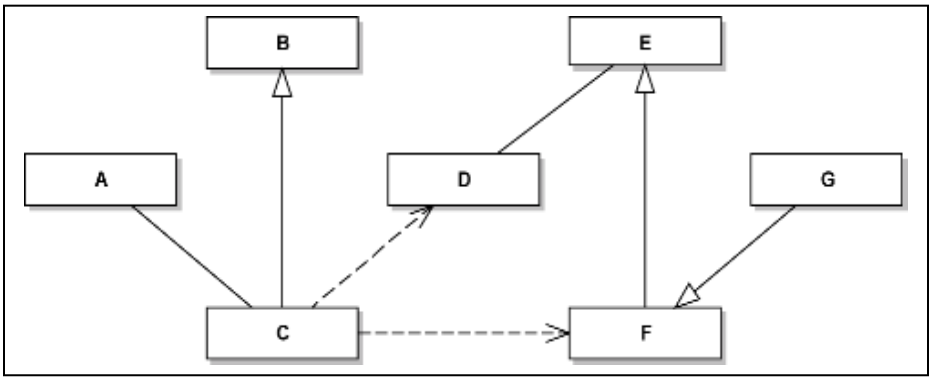


Figure 6.12: Assumed class model created from EA

The integrated concept of using multiple techniques for pattern recognition is novel in our approach. The numbers of techniques in the past have used these technologies for recognition of patterns as standalone, but no work is reported as combined effort. Some

approaches have used simple concept of SQL for recognition of patterns, but they are not able to extract all the information that realize the existence of patterns in the source code and report very low precision and recall. Our approach used extensive .Net framework and modeling tool EA which support multiple language pattern recognition. It can be extended for other languages by using the same feature types and queries with additional effort of implementing source code parsers for new language. We extracted very detailed information about objects playing role in pattern definitions which help program maintenance and comprehension. The extracted information can be further visualized for detecting overlapping and composition between design patterns.

6.4 Discussion

The ultimate worth of any pattern detection technique can be measured by detecting different structural as well as implementation variants of design patterns accurately. We discussed the limitations of different techniques in Chapter 3 and developed our technique to overcome the problems mentioned in different approaches. Some techniques claim the detection of design patterns from different software systems with 100% accuracy, but unfortunately extracted results have number of false positives and false negatives noticed during manual analysis of source code. It is also noticed that accuracy rate of different techniques decreases as the size of software used for pattern detection increases. For example, some approaches claim 100% accuracy on very small examples, but when same approach is tested on large scale systems, the approach either fails to detect patterns or results have very low accuracy. The definitions of design patterns are key concern for evaluating results of different tools. The increased numbers of structural as well as implementation variants of design patterns hamper the accuracy of different approaches. The flexibility and customization of pattern definitions and their translation into SQL/REGX/SCP features allow user to add a new feature or remove existing features from pattern definitions in order to detect different variants which improve accuracy of presented approach. Moreover, the approach is capable to detect patterns from multiple language source code due to support of Enterprise Architect Modeling Tool for reverse engineering source code and compatibility of parser module for each language. Currently, we have full parser module support for Java and C#. We used regular expressions for source code of C/C++, because source code parser support is yet under development. The source code parsers for C/C++ can be developed and integrated with the approach to detect dynamic features from source code of these languages more accurately. The multiple search methods and multi-language support with customizable pattern definitions used for pattern detection make our approach novel and unique. Finally, the approach can be extended to detect overlapping and composition of different patterns.

6.5 Summary

This chapter discussed pattern recognition process used for the detection of design patterns from the legacy source code of different applications. The evaluation of recognition process is presented in Chapter 8. It discussed the objectives and scope of approach, multiple techniques used in pattern recognition process and architecture of approach. The objectives and scope of approach used for pattern recognition process is discussed in Section 6.1. Section 6.2 discussed the multiple techniques which support the recognition process described in Section 6.3. Section 6.3 discussed the architecture and application of approach used for pattern detection. The static and dynamic overview of pattern recognition process and steps used to detect patterns are illustrated. Pattern recognition process is illustrated with example. Finally, Section 6.4 discussed the major features and effectiveness of the approach used for pattern detection.

Chapter 7

Design Pattern Recognition Prototype

This chapter describes a prototyping tool called UDDPD (User Defined Design Pattern Detector), which is implemented based on the methodology discussed in the previous chapter. The current implementation of tool focuses on all the GoF patterns and user defined patterns. Section 7.1 discusses goals that are formulated for the development and the implementation of tool. The components used for pattern extraction are discussed in Section 7.2. It further describes the architecture and implementation of the prototype. Section 7.3 explains the algorithms used for the implementation of prototype. The features of prototype are discussed in Section 7.4. The comparison of prototyping tool with other tools is presented in Section 7.5. Finally, Section 7.6 discusses the conclusions, shortcomings and constraints that address the loose holes for the future work.

7.1 Goals for Prototype Tool

Design pattern recovery techniques are assisted by different tools to validate concepts of techniques used for pattern detection. Each tool is developed to support the particular methodology and it is difficult to integrate a tool with other tools. Design pattern research community has also paid little attention towards development of open source tools for design pattern recovery. The detailed review about different design pattern recovery tools is presented in Section 3.2. Here we mention different approaches used to evaluate and compare different design pattern recovery tools which give us guidelines for setting goals for our prototyping tool.

Gueheneuc et al. [112] have presented a comparative framework for design recovery tools on the basis of parameters; the context in which it is applied, its Intent, its Users, its Input and Output, the technique which it implements, its actual Implementation, and the tool itself. Authors compare their self developed tool Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java [141]) with LiCoR (Library for Code Reasoning [142]) on the basis of eight mentioned factors. Similarly, Fulop et al. [137] have presented a benchmark for evaluating design pattern mining tools. The benchmark is general regarding programming languages and currently contains results of three tools (Columbus (C++), Maisa (C++), DPD (Java)).

The efficiency and completeness of a pattern matching tool is determined by accurately matching different pattern implementation variants. Most design pattern recovery tools have little or no support for documenting the presence and usage of patterns in source code. Some tools are language dependent and require complete source code without syntax errors. The tools differ in matching algorithms, pattern descriptions, pattern representations, precision, recall, etc. Based on the literature review about different design pattern recovery tools presented in Section 3.2 and objectives of our approach, we formulated the following goals for our prototyping tool.

1. The tool should take structural, dynamic and semantic features of patterns as input and it should match the features with source code elements participating in the pattern structure.
2. The tool should be flexible enough that it can detect different variants of a pattern by allowing user to customize features of patterns.
3. The tool should be scalable that it can detect patterns from source code of legacy applications ranging from small to million lines of source code.
4. The tool should be able to extract patterns from source code of multiple languages.
5. The output presentation must be applicable to maintenance and comprehension.
6. The tool should detect full catalogue of GoF patterns and user defined patterns with good precision and recall from different legacy applications.

7.2 Concepts, Architecture and Implementation of Prototype

The basic concept for the development of prototyping tool is based on the feature types which are used to describe characteristics of patterns. The combinations of related feature types realize the existence of a pattern in the source code. Each feature type is translated into SQL query or regular expression or parser module feature.

The prototyping tool is implemented based on the concepts and the techniques discussed in Chapters 4, 5 and 6. It has been implemented using Microsoft Visual Studio.Net framework by creating Add-In with Sparx System Enterprise Architect Modeling Tool. It supports design pattern detection from source code of multiple languages. The architecture of prototyping tool is shown in Figure 7.1. It consists of three modules namely: data, code and presentation modules. The data module contains pattern definitions, feature types, SQL queries, source code parser features and regular expressions. The artifacts in this module are independent from other modules and customizable only with the exception of source code parsers. For example, if user wants to change a feature type or a pattern definition, he/she does not need to dig into the source code details for changes. The parser module features are also independent in data module, but changes in the parser module features require analysis of code module.

The code module consists of application programs written in C# integrated with Enterprise Architect Modeling Tool as Add-In. Code module takes input from data module and use pattern recognition process to recognize different instances of patterns. This module is used for creating environment for pattern matching. The pattern matcher uses facts from the data module to recognize different instances of design patterns.

The presentation module is used to represent the recovered pattern results. Most of the pattern matching tools just give information about the presence of a pattern in the source code and provide no idea about exact location of participating classes in the pattern and their multiple roles in the other patterns. The presentation of results extracted by pattern detection approach is very important for maintenance and comprehension of legacy source code.

We already developed a custom built tool DRT (Design Recovery Tool [108]) to extract different artifacts from the source code of legacy applications. DRT used regular expressions for pattern matching and have good library of different regular expression patterns, which we applied in [109] for extraction of different artifacts. The pattern specifications of DRT are abstract and can be used to detect different artifacts from source code of multiple languages. We take the advantage of using regular expression patterns of DRT in our new prototyping tool, because the Visual Studio.Net framework supports parsing of regular expressions. The library of regular expression patterns is stored in a text file and it is used by our prototyping tool. These pattern definitions can be iteratively used

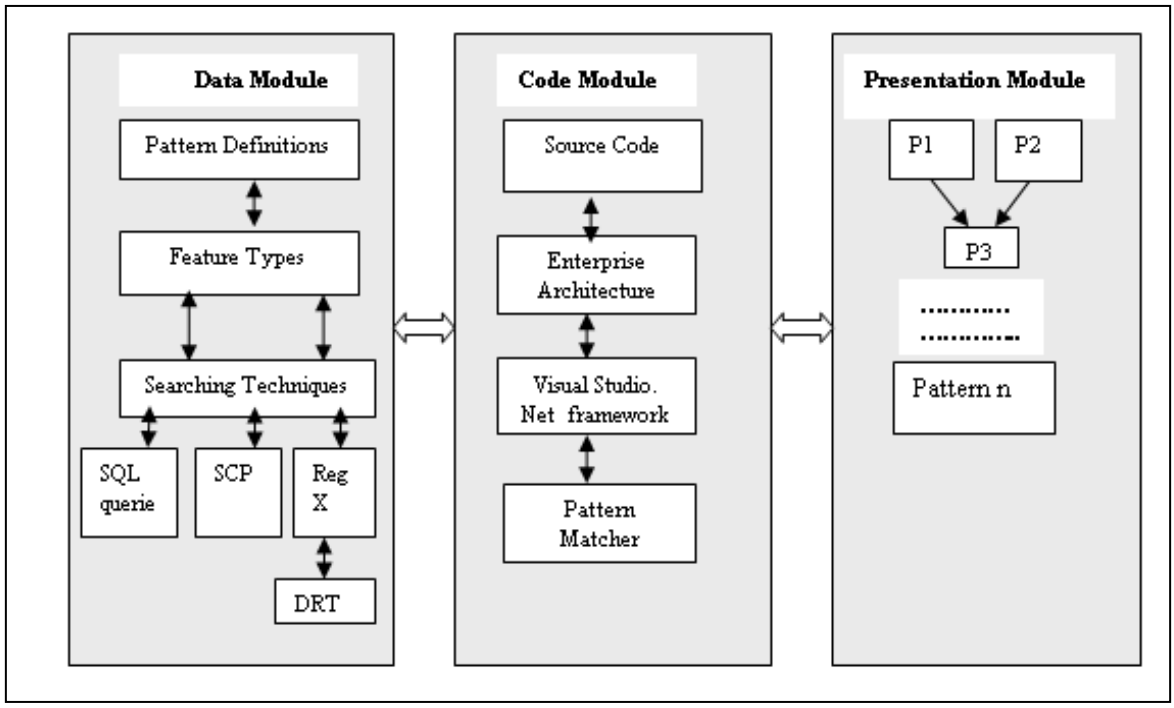


Figure 7.1: Architecture of prototyping tool

and may be refined according to the requirement of matching varying features of different patterns. The combination of regular expressions, SQL queries, source code parsers and semantics of annotations make our tool unique for fast pattern extraction.

The pattern detection process followed by the prototyping tool is explained in Figure 6.11 with the help of an example. It requires prior step of reverse engineering source code of legacy applications and creation of source code model. Source code model gave us opportunity to analyze intermediate representation of source code as well as directly use source code for extraction of patterns. Source code parsers take source code as input and extract relevant features from the source code elements which are necessary to match instances of different design patterns. The tool can use annotations for the candidate classes, which are used for the detection of patterns. This reduces the search space and time by taking only important interfaces, classes and methods for any design pattern. The detection of accurate implementation variants of design patterns determines the capability of a tool. Most of the tools result many false positives, because they do not implement all the variations of a design patterns in their tool. For example, Factory method has nine structural and implementation variants as mentioned in Chapter 2, and it becomes important for the tool developers to consider all the variations in the implementation of tool. The formalization of design pattern definitions can specify the variants of different design patterns.

7.3 Algorithms for Pattern Recovery

Different approaches use different algorithms for pattern recovery due to various implementation variants of a single design pattern. The numbers of approaches get the intermediate representations from the source code which affect the algorithms for pattern recovery. We have analyzed the algorithms used in [3 19 25]. The algorithms are hard coded in the source code and fail to detect the variants of different patterns. Our approach takes the advantage of getting intermediate representation of source code from Enterprise Architect Modeling Tool [26] in the form of source code model. The missing information in the intermediate model is extracted directly from source code using parser module features and regular expression based pattern definitions. The used annotations reduce search space and time for detecting different patterns. For example, while extracting Proxy pattern, we extract all the proxy and real subject classes based on annotations and then explore the relationship between pairs of proxy, real subject and subject classes.

Pattern recognition approach used the combinations of annotations, SQL queries, parser module features and regular expressions to match with varying features of design patterns. The SQL queries and regular expression patterns are customizable and not hard coded in the source code. The sample pseudocode for Proxy pattern detection is shown in Figure 7.2. The algorithm used for extracting delegation information is mentioned in

Figure 6.7. The variants of Proxy can be matched just by selecting the alternative features from feature type list. However, it is worthwhile to mention that annotations are optional for pattern detection approach.

I) Select Rules/Features of Pattern(for example Proxy)
II) Translate Features into following SQL queries and Regular expressions.
III)
Q1: Find all proxy classes based on annotations.
Q2: For all Proxy Classes, check their interfaces. (Generalization or Realization) with subject classes)
Q3: For all Proxy Classes, Check the associations with real subject classes.
Q4: For each Proxy, subject and real subject classes, search the common request methods.
R1: Check the delegation of request methods in Proxy and real subject.
IV) Present the results.

Figure 7.2: Pseudo code for Proxy pattern detection [111]

7.4 Features of Prototyping Tool

The prototyping tool is developed on the basis of goals articulated in Section 7.1. It has following fundamental features which make it unique from the other tools available in the market.

Input and Output

Input for tool are the feature types related with each pattern. The combinations of feature types are used to define the properties of patterns. The alternative features can be selected by the user to match variants of patterns. The user defined variable feature types can match all GoF design patterns. The visualization and animations techniques are not only important for program comprehension, but they also play key role for the presentation of extracted design pattern instances. The extracted information cannot be very useful until, it is presented in understandable format. The large and complex systems having the numbers of artifacts and different kind of relationships between the artifacts pose challenges for the presentation of patterns results. The tool dominates from the other tools, because it present extracted pattern information by indicating the actual location of a pattern's instances in the source code files. For example, the delegation parser extracts line numbers in the source code where delegation call is invoked. See Figure 7.3 which shows delegation call extracted by the delegation parser to recover different instances of

Figure 7.3: Extracted patterns from Junit3.7

Adapter pattern. We preferred to present the extracted pattern information by indicating the actual location of pattern instances in the source code files which is very important for the maintenance and comprehension of source code artifacts. The extracted information about Adapter pattern from Junit3.7 is shown in Figure 7.3.

Flexibility and Customization

The prototyping tool is implemented as modular tool that it is composed from different components and changes in one component has no affect on other components. For example, the SQL queries are stored in separate file and user can add, remove and modify queries without going into pattern detection module. Similarly, feature types use static and dynamic parameters to integrate the results of different searching techniques. The integration of multiple techniques adopted by our prototype make it flexible enough to detect user defined pattern variants. The inheritance features of Visual Studio.Net framework and Enterprise Architect Modeling Tool are flexible for the extension of prototype features, which can be used for the analysis and the presentation of extracted results. One major goal of developing prototyping tool was to support user that it should reuse and customize the feature types and the pattern definitions. This allows user to refine feature types, searching technologies and pattern definitions for matching the variants of design patterns. Most of the tools use hard coded algorithms which make difficult/impossible for the user to make changes in the source code. The pattern matching capability of tool can be extended by designing new feature types, pattern definitions and algorithms to recover broader range of patterns.

Abstraction and Extraction

Most design pattern detection tools extract and abstract artifacts in different styles at different levels, which are not very much relevant to maintenance task and design pattern recovery. The abstraction makes it possible to customize the pattern definitions. The extraction of artifacts in our prototype do not depends only on the database information, because the parser module can extract information directly from the source code. Furthermore, the extraction and abstraction operations may be refined by the user according to its need.

Performance and Scalability

The performance and scalability of pattern matching tools is also very important attribute, because the legacy system under study may have million lines of code and it may not be structured that whole search operation can be performed on subset of code. It indicates how well a system should complete its tasks (speed, accuracy and the resources it takes). The independent modules of our prototype can use the subsets of systems to extract the different patterns which resolve the issue of performance. Although, we focus on accuracy of pattern detection and performance of tool was not the major concern. The implemented tool is scalable, because it is tested on large open source applications as mentioned in Chapter 8.

Language Independence and Portability

Most design pattern recovery tools are language dependent and they do not support multiple languages for pattern recovery. Prototyping tool used in our approach is language independent. Currently, it supports Java, C#, C/C++, but it can be extended for languages supported by Enterprise Architect Modeling Tool. It requires only additional effort of developing new source code parsers for each new language. Portability is key feature of any pattern detection tool. The used prototyping tool is language and software independent. It is implemented using .NET framework which can integrate different capabilities of this framework**.** It can be extended for different languages that are supported by the Enterprise Architect Modeling Tool.

Precision&Recall

The tool should have ability to match the required patterns with accuracy and it should have good precision and recall rates. The prototyping tool used in this approach is tested on different open source systems and results are validated manually with other tools to ensure the correctness and completeness. The combined effect of precision and recall calculated as F-Score in Section 8.5 show the accuracy of prototyping tool. However, it is necessary to revalidate the tool on different industrial and commercial applications.

7.5 Comparison and Evaluation of the Prototype

The numbers of design pattern recovery tools are used by industry and academia and it was essential for us to compare/evaluate features of existing tools before development of our prototyping tool. It was necessary to collect, organize and analyze several sources of information related with pattern recovery tools for their evaluation. We reviewed features of selected tools discussed in Section 3.2 and found that most of the tools are developed to support the particular methodology. It is obvious from analysis of related work that little attention is paid on developing tools which should be integrated with other tools. The review about features of design pattern recovery tools is presented in Chapter 3 and features of our tool can be compared with the tools as mentioned in Table 3.3.

Commercial, academic and open source tools exist in literature, but the important question is to get accurate information about these tools. The code level and model level support have been noticed in different tools used for pattern recognition. The code level tools directly take the source code as input and use different techniques (AST, ASG, RSF, PDG, etc.) to represent the source code. These tools use different algorithms to extract patterns from different representations of source code. The tools report more errors because the hard coded algorithms used by these tools are not capable to detect the implementation variants of design patterns. The model level tools such as (IBM Rational Rose[144] and Enterprise Architect Modeling tool [26], etc.) are used to create the intermediate representation of the source code in the form of different models and then use different techniques to extract patterns from the data available in the models. The accuracy of these tools depends on the ability of modeling tools. To the best of our knowledge, most of the modeling tools are still not able to extract all relationships from the source code. The both level of tools have their strengths and limitations. We used the integrated application of code level and model level support for implementation of prototyping tool.

The customizable pattern definitions have improved precision/recall of pattern extraction and multiple languages support for pattern extraction make our prototyping tool unique from the other tools. The output format can be used by different modeling tools for the visualization of extracted results. The detailed information extracted about existence of patterns can be used for program maintenance and comprehension. We evaluated precision, recall, F-Score and scalability of tool by performing experiments on different open source examples mentioned in next chapter.

7.6 Discussion and Summary

The prototyping tool is flexible, scalable, have comprehensible I/O formats, abstraction support and good precision/recalls rates. It is developed as Add-IN with Enterprise Architect Modeling Tool which has very good reverse engineering capabilities for more

than ten programming languages. It can be used as Add-IN with other Modeling tools such as IBM Rational Rose, Code Logic, etc. Prototype can be extended by creating a UML profile for each feature type used for pattern description. Further, the UML pattern structure can be transformed into SQL queries and regular expressions. The additional effort for extraction of patterns from source code of other languages is the development of a parser module for each new language.

The goals of tool are discussed in Section 7.1. Section 7.2 described the concepts, architecture and implementation of tool. The algorithms used for the implementation of tool are discussed in Section 7.3. The features of implemented tool are discussed in Section 7.4. Section 7.5 compared and evaluated the features of our prototyping tool with other tools as mentioned in Chapter 3.

Chapter 8

Evaluation

This Chapter presents evaluation of the approach and the tool used to detect instances of design patterns from the source code of different applications. The scalability of the approach is validated by performing experiments on the systems ranging from small examples to very large scale open source systems. Section 8.1 discusses the considerations and assumptions taken for evaluation of results presented in this thesis for correctness and completeness. Section 8.2 discusses the setup of examples and benchmarks selected to perform experiments. The extracted results are presented in Section 8.3. The presented results are discussed and compared with the baseline approaches. Section 8.4 discusses precision, recall and F-Score of results extracted by our approach. The discussion about disparity of results is discussed and presented in Section 8.5. Section 8.6 analyses the threats to the validity of extracted results. Finally, Section 8.7 summarizes the evaluation results and discusses their implications.

8.1 Considerations and Assumptions for the Evaluations

We evaluated our approach using precision, recall, F-score [67] as explained in Sections 8.3 and 8.4. Experiments are performed on setup of examples mentioned in next section. It is worthwhile to clarify that results extracted by approach used in this thesis are based on some considerations and assumptions. The selection of trustable benchmarks examples was key challenge for comparing and evaluating our results. We selected strong, moderate and weak baselines due to justifications given in Section 8.4. We take assumptions on the results of baselines approaches, because they claim manual analysis of their results and believe that their extracted results are 100% accurate. Unfortunately, some discrepancies are observed in the results of these approaches. For small systems, it is easy to analyze manually extracted results, but it is not feasible and quite time consuming for very large size of systems. Even if two approaches extract the same pattern results, it still remains questionable that how accurate results are until manual analysis of source code is not performed. We accepted and took assumption on the results of well known approaches as baselines to evaluate our approach. Another problem arises because the approaches that we take as baselines do not extract all the patterns from the source code. So it becomes more serious concern for patterns which have no previous results. We analyzed

our extracted results manually and validated them according to our pattern definitions, but it was not in our scope to analyze all the source code manually to validate the false negatives. Further research is required from the design pattern community on the benchmark systems to fill this gap. Similarly, our results are based on the assumption of annotations in the source code for patterns (State, Strategy, etc.) which cannot be extracted with structural and behavioral analysis. Finally, we tried our best to define the alternative pattern definitions to extract the possible variants of different patterns, but the implementation variants depend on the styles of programmers. Due to everyday new breakthrough in the features of programming languages, the implementation of patterns become easy, but it poses challenges for the existing approaches to detect new variants of patterns accurately.

8.2 Experimentation Setup

Experimental setup is the collection of related source code examples to perform experiments for the validation and the scalability of presented approach. Experiments are performed on different software systems having source codes in different programming languages. The extraction of patterns from source code of multiple languages makes our approach unique, but it poses challenges of developing parsers for each new language. Some approaches performed experiments on very small examples to validate their results, but they face the problem of scalability in the case of larger systems. The scalability and performance of pattern extraction approaches is also important concern. We selected sample systems for experiments due to the following fundamentals:

I) These systems are selected because the most successful approaches have performed experiments on one or more of these systems and we can compare our results with other approaches. While analyzing results of these approaches during our literature review, we observed a wide disparity in the results of most approaches using these examples which motivated us to further investigate the causes of disparity in the results.

II) Secondly, the source code of these applications is available freely to perform experiments.

III) Thirdly, the size of these systems varies from few lines of source code to over a million lines of code, which is important to validate the scalability of our approach.

IV) We selected source code of different languages (Java, C/C++ and C#) for validating the generalization of our approach for multiple languages.

V) Finally, most of these applications have been developed by using different design patterns.

The following open source systems/examples of (Java source code) are selected on the basis of previous mentioned arguments:

I) AJP [122].
II) JUnit 3.7[129].
III) JHotDraw 5.1 [130].
IV) JRefactory 2.6.24 [131].
V) QuickUML 2001 [132].
VI) Apache Ant 1.6.2 [133].

Table 8.1: Java source code statistics

	JHotDraw 5.1	**Junit 3.7**	**JRefactory 2.6.24**	**QuickUML 2001**	**Apache Ant 1.6.2**
Availability	www.jhotdraw.org	www.junit.org	www.jrefactory.sourceforge.net	www.sourceforge.net/projects/quj/	www.ant.apache.org
Size on disk	4.85 MB	1.46 MB	12.5 MB	3.39 MB	26.6 MB
Code files	144	78	1167	152	259
LOSC	30860	9742	216244	46572	72.4KLOC
LOCs	454	126	4110	2128	55543
BL	4896	1074	28047	11692	28343
Packages	16	9	58	13	76
Classes	136	43	562	204	883
Interfaces	19	9	13	12	66
Methods	1314	425	4881	1082	8272
Attributes	331	114	1367	422	4388
Associations	74	16	131	70	216
Generalizations	103	12	327	83	566
Realizations	26	13	37	24	152
Total connections	203	41	495	177	934

LOSC: Lines of source code **LOCs: Lines of Comments** **BL: Blank lines**

The information in Table 8.1 is extracted using Enterprise Architect Modeling Tool and regular expression patterns. JHotDraw 5.1 is a Java framework for drawing two-dimensional technical and structural graphics. It is commonly used to develop and customize graphical editors for different applications. It includes several examples of editors, in particular a simple one to draw color rectangles, circles, and texts. This framework is developed by using different design patterns. So it is the most appropriate example for performing experiments on design patterns. Junit 3.7 is a unit testing framework for java applications, which is developed to ease the implementation and running of unit tests for Java applications. It decouples the test inputs from test implementation which makes test inputs useable for many test implementations. JRefactory 2.6.24 is a tool that can perform different refactoring operations on Java source files and update files after refactoring. It can be used as command line with GUI and has been integrated in various IDEs, including Sun's NetBeans, JBuilder, JEdit, etc.

QuickUML 2001 is an object-oriented design tool that supports the design of a core set of UML models and has advanced features like design namespaces for project organizations. It can be used to customize UML projects using stereotypes, color schemes, etc. Finally, Apache Ant 1.6.2 is an open source software tool used for automating build software processes. It can be considered as a kind of application like make utility, but it is written in Java and is primarily intended for use with Java [133]. All of these applications are developed by using one or more design patterns.

The sizes of open source systems range from 10, 000 lines of code (LOC) to about 72.4 KLOC with a number of classes vary in size from 43 to 883 classes. The information about major artifacts of source code examples is given in Table 8.1. We include only .java files to calculate the number of code files in counting.

We also evaluated our approach on different systems/examples developed in C/C++ language as given below.

I) Huston [122].
II) Galb++ [62].
III) Libg++ [63].
IV) Mec [134].
V) Socket [135].

Table 8.2: C/C++ Source code statistics

	Galb++(2.4)	**Socket(1.10)**	**Mec(0.3)**	**Libg++(2.7.2)**
Size on disk	1.17 MB	644KB	1.37MB	8.86 MB
Code files	135	49	65	99
LOC	20,507	3078	21,006	44.106
LOCs	1528	411	2855	3563
BL	2111	1258	4198	7388
Packages	16	4	3	2
Classes	98	30	132	208
Methods	1128	110	246	2000
Attributes	431	5	529	674
Associations	22/40	6	11	84/96
Generalizations	51	19	6	60
Realizations	0	0	0	0
Total connections	91	0	290	156

The information about artifacts of these systems is mentioned in Table 8.2. The information is extracted using Enterprise Architect Molding Tool and regular expression patterns. Galib++ [62] is a C++ Genetic Algorithm Library used to solve optimization problems. It is available freely for experiments. Mec [134] is a trace-and-replay program used to review program execution exactly through traces. Socket [135], which is a library

for inter-process communication, and Libg++ [63], which is part of the GNU Free Software foundation C++ development environment.

Finally, examples/open source systems for performing experiments on source code of C# were not available. To-date, all the authors have performed experiments on source code of C/C+++/Java/Smalltalk. Our approach is capable to detect patterns from multiple languages and currently it can detect patterns from C# source code. We have only performed experiments on GoF implemented C# source code examples [123] due to unavailability of open source examples and commercial applications. The feature types and parser modules for analysis C# source code are implemented in our prototyping tool.

8.3 Experimental Basics and Results

We evaluated and compared results of presented approach step by step in this section as follows:

I) Firstly, we performed a pilot study on source code of GoF implemented examples including (Java, C/C++ and C#). The sample source codes for single pattern implementations are taken from [121 122 123] respectively for each language. The AJP book [121] provided a list of examples of Java patterns along with the description of each pattern, which therefore immediately provided a source of good examples that could be used to test our prototype. The source code for each pattern is available for download. The design pattern framework 3.5 book [123] contains implementation of patterns in C# with simple and real world examples. The source code was available for validation of our pattern definitions. The prototype detected patterns from C# source code using same feature types. Finally, Huston GoF design patterns by famous authors of GoF book [122] is excellent online source which contain pattern examples implemented in C++ as well as in Java. The source code for examples of C++ implemented GoF patterns is selected from this resource. The regular expression patterns are used to extract features which are extracted using source code parsers for Java and C#. However, the accuracy of extracting information using regular expressions is low as compared with source code parsers.

We reverse engineered all the GoF implemented source code examples for each language randomly using Enterprise Architect Modeling Tool in one file and used our pattern detection technique as initial experiment. This step validated features types and pattern definitions used for the recognition of patterns. We experienced overlapping in the definitions of some patterns (State/Strategy and Adapter/Command) in this experiment. After validation of single pattern definitions, the approach is used to analyze and detect patterns from different open source legacy applications ranging in size from small, medium to large size systems. The approach extracted patterns from source code of different languages (Java, C/C++ and C#) with the help of different source code parsers used for each language. The novice user can customize pattern definitions which are not

hard coded in the source code. The approach detected all the GoF patterns from source code of these sources successfully.

Table 8.3: Multiple language pattern detection

Pattern	AJP(Java)	C#	C++(Huston)
Singleton	++	++	++
Factory Method	++	++	++
Abstract Factory	++	++	-+
Prototype	-+	++	-+
Adapter	++	++	++
Proxy	++	++	++
Decorator	++	++	++
Bridge	++	++	++
Composite	++	++	++
Template Method	++	++	++
Observer	++	++	++
Visitor	++	++	++
Flyweight	++	++	++
Mediator	-+	++	++
Builder	++	++	++
State/Strategy	-+	+-	-+
COR	++	++	++
Interpreter	-+	-+	-+
Command	-+	-+	-+
Memento	-+	-+	-+
Iterator	-+	-+	-+
Facade	-+	-+	-+

++: Tool detect pattern with all True Positives
-+: Tool detect pattern with some false positives
--: Tool detects pattern as false negatives

II) The second set of experiment is performed on different examples as mentioned in Table 8.1. The selection of examples is due to good baseline results. We started our initial test on Junit 3.7 which is relatively small example and recovered 98% same precision and 100% recall as detected by baseline approaches [20 45]. In the continuation, the approach is tested on other examples and it extracted improved precision and recall in comparison with baseline approaches. Each cell in Table 8.4 gives information about six facts related with one pattern instance. The first three cells in the row show our extracted results, precision and recall of extracted results. The below three cells show the results of three baselines. The "0" in any cell reflects that approach has not detected any pattern and "x" shows that approach is not capable to extract this pattern. The cell having information like 6(6, 1) shows that baseline [20] shares "6" and baseline [45] shares "1" instance with

approach presented in this thesis. The recall is not computed in comparison with weak baselines.

Table 8.4: Extracted pattern instances

Software	**JUnit3.7**			**JHotDraw5.1**			**JRefactory 2.6.24**			**Quick UML 2001**			**Apache Ant1.6.2**		**AP &R**
Results	**[PI]**	**P**	**R**	**[PI]**	**P**	**R**	**[PI]**	**P**	**R**	**[PI]**	**P**	**R**	**[PI]**	**P**	
Reference	**[20]**	**[45]**	**[50]**	**[20]**	**[45]**	**[19]**	**[20]**	**[45]**	**[39]**	**[20]**	**[45]**	**[19]**	**[25]**	**[19]**	
Singleton	0(0,0)	1.0	1.0	2(2,2)	1.0	1.0	12(12,2)	1.0	1.0	1(1,1)	1.0	1.0	1(1)	1.0	1.0
	0	0	x	2	2	x	12	2	1	1	1	x	1	x	1.0
Adapter	6(6,0)	1.0	1.0	22(18,1)	1.0	1.0	16(16,16)	1.0	.72	10(10,0)	1.0	1.0	21(8)	.71	.94
	6	0	x	18	1	41	26	17	16	11	0	27	13	41	.93
Composite	1(1,1)	1.0	1.0	1(1,1)	1.0	1.0	0(0,0)	1.0	1.0	2(1,1)	1.0	1.0	5(2)	.60	.92
	1	1	1	1	1	0	0	0	x	1	2	0	44	4	1.0
Decorator	1(1,1)	1.0	1.0	3(3,1)	1.0	1.0	1(1,0)	1.0	1.0	0(0,0)	1.0	1.0	2(2)	1.0	1.0
	1	1	x	3	1	0	1	0	x	0	0	0	12	0	1.0
Factory Method	0(0,0)	1.0	1.0	3(3,3)	1.0	1.0	1(1,1)	1.0	1.0	0(0,0)	1.0	1.0	5(3)	.60	.92
	0	0	x	3	3	0	1	1	0	0	0	x	6	x	1.0
Template Method	1(1,0)	1.0	1.0	5(5,2)	1.0	1.0	17(17,0)	1.0	1.0	4(4,0)	1.0	1.0	17(4)	.82	.96
	1	0	1	5	2	x	17	0	x	5	0	x	4	x	1.0
Prototype	0(0,0)	1.0	1.0	2(1, 2)	1.0	1.0	1 (0,0)	1.0	1.0	1(1,0)	1.0	1.0	6	.50	.90
	0	0	x	1	2	x	0	0	x	7	0	x	x	x	1.0
Command	0(0,0)	1.0	1.0	8(8,1)	1.0	1.0	0(0,0)	1.0	1.0	2(0,0)	0	0	14	.71	.74
	0	0	x	8	1	x	0	0	0	0(0)	1	x	x	x	.75
Observer	2(1,1)	.50	1.0	2(2,2)	1.0	1.0	0(0,0)	1.0	1.0	0(0, 0)	1.0	0	4(3)	1.0	.90
	4	3	4	5	2	x	0	0	x	0	1	x	5	x	.75
Visitor	0(0,0)	1.0	1.0	0(0,0)	1.0	1.0	2(2,2)	1.0	1.0	0(0,0)	1.0	1.0	2(1)	1.0	1.0
	0	0	0	0	0	x	2	2	2	0(0)	0	x	1	x	1.0
State/Strategy	3(3,0)	1.0	1.0	20(20,6)	.95	1.0	8(8,2)	.73	1.0	2(2,0)	1.0	1.0	19	.73	.88
	3	0	x	22	6	x	11	2	3	15	0	x	26	x	1.0
Proxy	0(-,0)	1.0	1.0	0(-,-,0)	1.0	1.0	9	1.0	nbl	1(-,-,1)	1.0	1.0	3(3)	1.0	1.0
	x	x	x	x	x	0	x	x	x	x	x	1	27	0	1.0
Bridge	0	1.0	nbl	5	1.0	nbl	0	1.0	nbl	0(-,-,0)	1.0	1.0	9	.55	.91
	x	x	x	x	x	75	x	x	x	x	x	22	5	25	1.0
Interpreter	1	1.0	nbl	8	1.0	nbl	1	1.0	nbl	1	1.0	nbl	0	1.0	1.0
	x	x	x	x	x	9	x	x	x	x	x	x	79	x	-
Builder	0(-,0)	1.0	1.0	2(-,0)	0	1.0	5	?	nbl	0	1.0	0	0	1.0	?
	x	0	x	x	0	x	x	2	x	x	1	x	x	x	.75
Iterator	0	1.0	nbl	0	1.0	nbl	2	1.0	nbl	4	1.0	nbl	1	1.0	1.0
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-
Memento	5	1.0	nbl	10	1.0	nbl	30	1.0	nbl	5	1.0	nbl	43	.70	.94
	x	x	x	x	x	x	x	x	x	x	x	x	x	x	-
COR	0	1.0	nbl	0	1.0	nbl	1	1.0	nbl	0	1.0	nbl	3(2)	.67	.93
	x	x	x	x	x	x	x	x	x	x	x	x	3	x	-
Abstract Factory	0(-,0)	1.0	1.0	0(-,0)	1.0	1.0	0(-,0)	1.0	1.0	0(-,0)	1.0	0	0	1.0	1.0
	x	0	x	x	0	x	x	0	x	x	2	x	6	x	.75
Flyweight	3	1.0	nbl	15	1.0	nbl	15	1.0	nbl	2	1.0	nbl	7	.29	.85
	x	x	x	x	x	x	x	x	x	x	x	x	6	x	-
Facade	6	1.0	nbl	30	-	-	24	-	-	18(-,-,6)	1.0	wbl	47	.31	.46
	x	x	x	x	x	9	x	x	x	x	x	16	79	55	-
Average P&R		.98	1.0		.89	1.0		.98	.97		.95	.68		.77	

x: Means approach do not detect this pattern COR: Chain of Responsibility nbl: No Baseline wbl:Weak Baseline AP&R: Average Precision and Recall

III) The third set of experiments is performed on source code of C/C++ examples. The approach extracted structural patterns from C/C++ baseline source code samples as shown in Table 8.5. We did not perform experiments on other patterns due to unavailability of baseline results and parser module support for C/C++. The patterns are recognized on the basis of structural information extracted from Enterprise Architect Modeling Tool. The

delegation call information is realized with the help of regular expression pattern matching. We relaxed some constraints for detection of these patterns due to missing capabilities of modeling tool for reverse engineering C/C++ source code. For examples, the inline functions within type definitions are not reverse engineered by the used modeling tool.

Finally, the approach is capable to extract patterns from source code of C#, because the parsers for matching features of this language are developed and implemented. We validated our pattern definitions using same features on GoF source code examples only because we cannot find the open source systems for experiments on C# source code.

We used pattern recovery approach and tool to present results extracted from different applications in this section. The extracted results are compared with the other recent and successful pattern recovery approaches. The wide disparity in the results is visible from Table 8.4. We argue that our pattern recognition approach has extracted exact location of classes, functions and other artifacts participating in a pattern instance which is quite useful for the maintenance and program comprehension. For example, source code parser module extracted information about the exact line in the source code where delegation call is requested from the source object. The approaches like [23 35 29 50] extract only the number of patterns without exact location of pattern roles in the source code. The results of such approaches cannot be used for further analysis and maintenance of legacy applications. The approach is capable to recover complete information about the structure of individual pattern with the help of feature types. Each feature type extracts relevant information for desired properties of a design pattern. The common instances extracted from different approaches and the causes of disparity are discussed in Section 8.5.

Table 8.5: Extracted pattern instances from C/C++

System	**Galb++ 2.4**			**Libg++ 2.7.2**			**Mec 0.3**			**Socket 1.10**		
Reference	**[29]**	**[61]**	**[PI]**	**[29]**	**[61]**	**[PI]**	**[29]**	**[61]**	**[PI]**	**[29]**	**[61]**	**[PI]**
Adapter	6	4	5	0	0	0	1	1	1	1	1	1
Bridge	0	0	0	1	3	1	0	0	0	0	1	0
Composite	0	0	0	0	0	0	0	0	0	0	0	0
Proxy	0	0	0	0	0	0	0	0	0	0	0	0
Decorator	0	0	0	12	12	12	0	0	0	0	0	0

PI: Patterns Identified by approach used in this thesis

8.4 Precision and Recall Metrics

The precision and recall metrics are used for the evaluation of information retrieval techniques including design pattern detection approaches. There is also a long tradition of evaluating the quality of systems by measuring both precision and recall, i.e. how many of the documents retrieved are relevant and how many of the relevant documents are retrieved [128]. Recall is especially problematic when the numbers of extracted patterns are large in numbers and the false negatives cannot be assured without trusted

benchmarks. The relationship between precision and recall metrics determines the accuracy of any approach. Ideally, precision should remain high as recall increases, but in practice this is difficult to achieve [124]. Precision and recall also depends on the type of analysis used by the pattern recovery approaches. The static pattern analysis techniques have low precision for patterns which have weak structural signatures. The combined static analysis and dynamic analysis improve precision at reasonable rate. The semantic analysis supplements the static and the dynamic analysis and improves the recall rate by detecting false negatives. Precision and recall for design pattern detection techniques is measured on the basis of the following parameters:

True Positives (TP): It means that pattern is correctly realized in the source code.
True Negatives (TN):.It means that pattern is not recognized and it is not implemented.
False Positives (FP): It means that pattern is recognized which is not true.
False Negatives (FN): It means that pattern is not recognized but pattern is implemented.

Table 8.6: Metrics for precision and recall

	Recognized	Implemented	Result	Precision	Recall
TP	Yes	Yes	Required	TP/(TP+FP)	TP/(TP+FN)
FP	Yes	No	Avoided		
TN	No	No	Required		
FN	No	Yes	Avoided		

Precision and recall are important for measuring the accuracy of pattern recovery approaches, but the integration of both factors yield combined effect. Peterson et al. [67] have suggested an integrated common factor for measuring precision and recall metrics for any pattern recovery approach as standard solution to use the weighted harmonic means of P and R(weighted F-Score). They define weighted F-Score F_w, w€ R as:

$$F_w = \frac{(1 + w^2)(PR)}{(w^2P + R)}$$

The highest F-Score is obtained if both precision and recall are high. The suggested value of w =2.28. For precision of 100% and recall of 50%, the value of F_w will be 61% and if precision is 50% and recall is 100% then F_w=72% .

Baselines Assumptions for Precision and Recall

The precision and recall in Table 8.4 are calculated on the basis of certain assumptions. It is relatively easy to measure the precision of information retrieval approaches by comparing the extracted results with the given data, but the recall is challenging and questionable. The recall metric cannot be calculated until baseline results or documentation of systems is available. Most approaches performed experiments on open source systems and the documentation for these systems is not available or is mostly

obsolete. The only option for comparing our results was the selection of trusted baselines. We selected the results of successful approaches as baselines for the comparisons. We classify baselines as strong, moderate and weak baselines. The quality of baseline's results is very important concern and there exists no criteria for the selection of baselines. We selected baselines on the basis of following factors:

I) The authors conducted manual analysis of source code and claim 100% recall.
II) The extracted results are available on web for validation and comparisons.
III) The tools used for validations are publicly available.
IV) The approaches update their results continuously through feedback from community.
V) Analysis of technique and its results.
VI) The approaches perform experiments on all type of GoF patterns.
VII) Selection of experimental examples.
VIII) Results are published in top quality and trustworthy conferences and journals.

We selected approaches as strong baselines which have above mentioned metrics. The approach [20] has all the metrics that we defined for selection of strong baselines. The moderate baselines also follow same criteria only with the exception that applied tool is not available publicly to validate their results on the other examples. Secondly, the precision extracted by these approaches is low than strong baseline approaches. We selected [45 66] as moderate baselines because these approaches have major focus on recall, but they have very low emphasis on precision. The baselines which have very low precision/recall and their tools are not available for experiments on large examples are selected as weak baselines. These baselines do not claim the manual analysis of source code and do not calculate any recall values. The approaches [6 16] are selected as weak baselines on the basis of their available metrics. We did not calculate recall of our results against weak baselines.

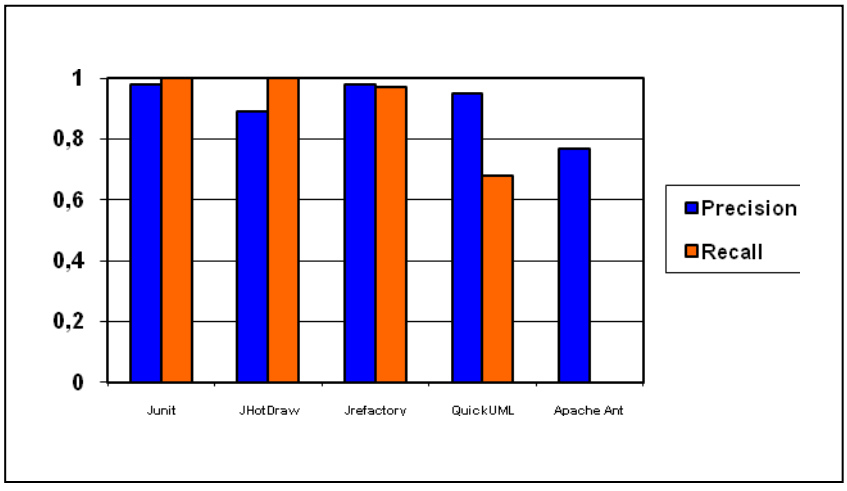


Figure 8.1: Precision and Recall of extracted results

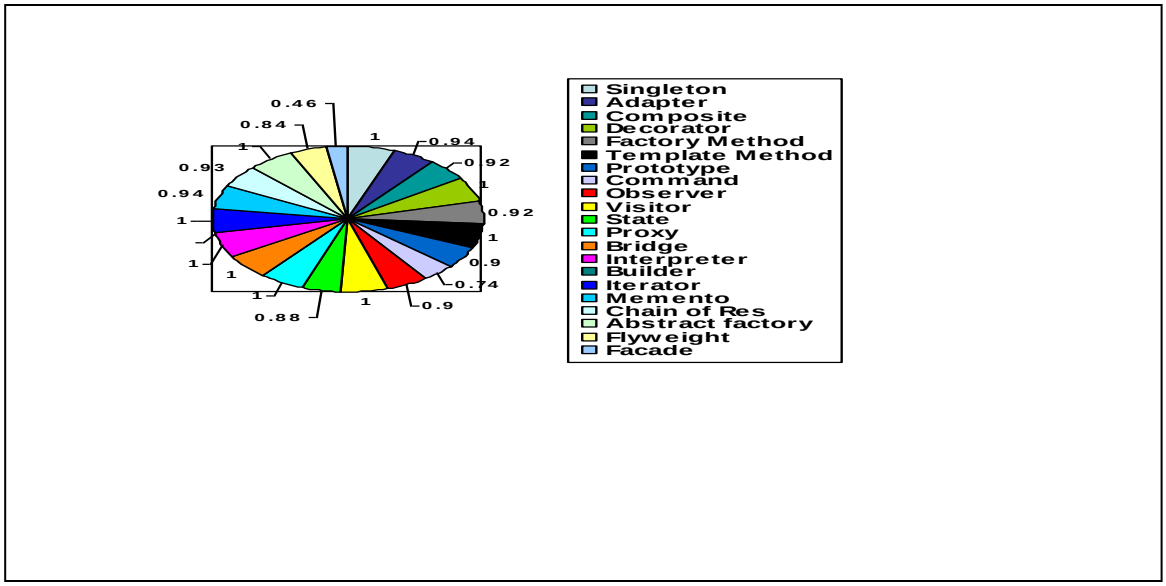


Figure 8.2: Extracted precision for each pattern

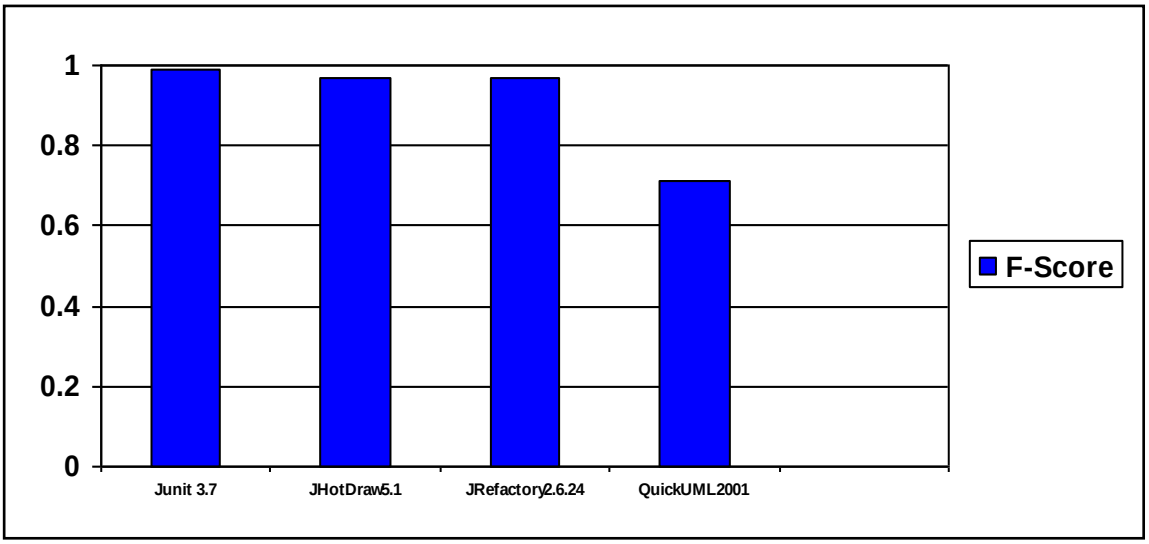


Figure 8.3: F-score including precision and recall

8.5 Disparity in Results of Different Approaches

The major causes for the disparity of results from different approaches are different definitions of same pattern, approximate/exact matches, unavailability of trusted benchmark systems/documentation, implementation variants, etc. Some approaches take standard GoF definitions of patterns and they do not consider different implementation variants of same pattern during pattern recovery. The standard GoF definitions have different interpretations by different authors which cause disparity in results extracted from different approaches. This wide disparity of results extracted from different approaches set some open issues that reflect the attention of design pattern and reverse

engineering community toward automated design pattern recovery. Reference [1] mentioned the causes of disparity in the results of different approaches and suggested benchmark systems for the validity of results. Different groups are working on the benchmark systems which will be used to compare the results of different approaches. To date, only benchmarks for few systems are available, which include only few patterns. We can circumvent the wide disparity by allowing users to customize the pattern definitions. The customizable pattern definitions can also detect the idioms and the elemental design patterns beyond the GoF patterns. Our prototyping tool is tested on different examples and we found large number of false positives, which are recognized as true positives by different approaches. We also observed few false negatives, which are not detected by different approaches. We cannot mention all the examples due to limitation of space and highlight only the following major disparities in the results of different approaches.

False Positives

I) Visitor pattern found
FileScanner is an abstract Visitor class
AbstractFileSet is a Vistee class
setupDirectoryScanner is the accept method
setBasedir is the visit method
dir is exposed to visitor FileScanner
File Location: src/main/org/apache/tools/ant/types/AbstractFileSet.java

PINOT [98] detected above as true Visitor pattern instance from Apache Ant 1.6.2, but our prototyping tool does not recognize this instance as true positive. Through manual analysis of source code, we found that this instance is false positive as validated by analyzing source code manually. There is reference of FileScanner class in the AbstractFileSet. The setupDirectoryScanner is recognized as accept method which calls the setBasedir visit method of FileScanner, but setBasedir has no reference of setupDirectoryScanner as argument in the FileScanner class. The method has argument of File Class as parameter. So it does not fulfill the definition of Visitor. Reference [20] also recognizes this as false positive, which further validates our finding.

Similarly in JHotdraw 6.0b1, PINOT [98] recognizes the following as correct Visitor instance.

II) Visitor pattern found
Storable is an abstract Visitor class
StorableOutput is a Vistee class
writeStorable is the accept method
write is the visit method
THIS pointer is exposed to visitor Storable

File Location: src/org/jhotdraw/util/StorableOutput.java

The storableOutput is recognized as concrete visitor class which should have parent class as interface with writeStorable as accept method. The object is the only parent class which does not have this accept method. Reference [1] also validates this as false positive.

Reference [20] recognized III given below as true positive from JHotdraw 5.1 as Adapter pattern. The approach recognized AbstractConnector as Adapter class and Figure as Adaptee class. We found that there is no delegation call from Adapter to Adaptee class, which is important condition for the validation of Adapter. Our approach filters this instance and does not recognize it as true Adapter instance.

III) CH.ifa.draw.standard.AbstractConnector
adaptee: Figure fOwner
Request (): displayBox, containsPoint
Target role is played by class CH.ifa.draw.framework.Connector

PINOT [98] detects IV given below as mediator from Apache Ant 1.6.2. Authors report PlainMailer as colleague class and Mailer as mediator class. According to GoF definition, there should be association from colleague to mediator. We did not find any association between these classes through manual analysis of source code.

IV) Mediator Pattern

Mediator: Mailer
Colleagues: PlainMailer, DateUtils
FileLocation: src/main/org/apache/tools/ant/taskdefs/email/Mailer.java

Similarly, we found another disparity in QuickUML2001 recognized by [19]. The authors have very strict criteria for pattern detection and they use static and dynamic analysis during pattern recovery process, but their approach has very diverse results in the case of some examples. The major cause of disparity in the results of this approach is the criteria for the detection of delegation information. This causes disparity in their results for the patterns which are using delegation.

V) Adapter Pattern
Quick UML 2001\src\uml\ui\PrintableAction
Quick UML 2001\src\uml\ui\ Diagramcontainer

PrintableAction and Diagramcontainer are recognized as Adapter and adaptee classes in [19]. While analyzing source code manually, we noticed that PrintableAction do not have association with Diagramcontainer. The Diagramcontainer instead contains reference of PrintableAction. The delegation condition is not satisfied to validate the correct instance for Adapter. Similarly, the authors report no Proxy pattern instance from Apache Ant 1.6.2, but PINOT [98] claim detection of 27 correct Proxy instances from the same

example. Through manual analysis, we realized that following instances as correct Proxy pattern instances. We mentioned only two instances due to limitation of space.

VI) Proxy Pattern
Available is a proxy
Task is a proxy interface
The real object(s): Path
File Location: src/main/org/apache/tools/ant/taskdefs/Available.java

VII) Proxy Pattern
PresentSelector is a proxy
BaseSelector is a proxy interface
The real object(s): Mapper
File Location: src/main/org/apache/tools/ant/types/selectors/PresentSelector.java

These numbers of false positives clearly threatens the validity of results extracted by these approaches and unfortunately the precision claimed by these approaches goes down. The numbers of false positives affect only precision, but the recall for some approaches is also affected when they are not able to recognize correct pattern instances.

False Negatives

The approaches [20 45] have not detected any instance of Prototype from JRefactory 2.6.24 as mentioned in Table 8.4 and they claim their recall 100% for Prototype in JRefactory 2.6.24. Our approach has detected the following instance of Prototype from this example, which deviate the claims of above approaches and their precision and recall become suspicious.

ASTName is the concrete prototype class and its clone method creates the copy of same class and returns this copy. The interface is provided by the cloneable. In spite of disparity in the results of [20], we still selected the approach as good baseline for comparing our results. For example, Table 8.7 shows the common instances extracted by [20] are also realized by our approach. Although, we have performed experiments on more patterns, but the baseline results are not available for comparing results of all patterns.

I) Observer Pattern (False Negative)
AbstractToolis is subject
ToolListner is observer
Toolstarted(), toolfinisned() are update methods

The above instance of Observer is not recognized by [20] from QuickUML 2001. We detected it as true Observer pattern, because it fulfills all the conditions of Observer.

Reference [45] has also detected this instance as true Observer which verifies our findings.

II) Adapter Pattern (False Negative)
CH.ifa.draw.contrib.PolygonTool.java
CH.ifa.draw.contrib.PolygonFigure.java

Our prototyping tool recognizes above as true Adapter pattern instance and we validated it manually by inspecting the source code. The PINOT [98] also validates our true detection by recognizing it as true Adapter instance. Similarly, reference [45] has also mentioned error in the result of above approach by not accurately recognizing Observer pattern from JHotDraw 5.1, which reduces the claimed recall. We validated through our prototyping tool and manual analysis of source code that III is not valid Observer pattern instance, because the MoveBy update method is not called within KeyPress, HandleCursorKey and MoveSelection methods.

III) Observer Pattern (False Negative)
CH.ifa.draw.framework.Figure
CH.ifa.draw.framework.FigureChange-Listener

The source code realizes that Figure and FigureChange-Listener play the roles of subject and observer classes.

IV)Adapter Pattern (False Negative)
CH.ifa.draw.standard.Commandattributefigure
CH.ifa.draw.framework.Figure

The strong and moderate baseline approaches [20 45] are not able to recognize above as adapter pattern instance. We detected this instance as true adapter pattern, which is validated through manual analysis of the source code. The approach [19] also recognized this as true adapter instance.

False negatives threaten the completeness of approach and are important for the maintenance of legacy applications. The moderate baseline [45] focused on achieving 100% recall by eliminating number of false negatives by scarifying precision. We focused on both precision and recall by using variant pattern definitions and tried to reduce the number of false positives and false negatives during pattern recovery process, which is the mantra of all design pattern recovery approaches.

8.5.1 Shared Pattern Instances

Table 8.4 compared the common instances of our approach with baseline approaches. Further, Table 8.7 shows the comparison of common instances of strong baseline with the approach presented in this thesis.

Table 8.7: Common instances shared with [1]

Software	**JHotDraw 5.1**			**Junit 3.7**			**JRefactory 2.6.24**			**Quick UML 2001**		
Reference	**[20]**	**[PI]**	**CI**	**[20]**	**[PI]**	**CI**	**[20]**	**[PI]**	**CI**	**[20]**	**[PI]**	**CI**
Singleton	2	2	2	0	0	0	12	12	12	1	1	1
Adapter	18	24	18	6	6	6	26	16	16	11	10	10
Composite	1	1	1	1	1	1	0	1	0	1	1	1
Decorator	3	3	1	1	1	1	1	0	0	0	0	0
Factory Method	3	3	3	0	0	0	4	4	4	0	0	0
Observer	5	2	2	1	3	1	0	3	0	0	0	0
Prototype	1	1	1	0	0	0	0	0	0	7	0	0
Command	x	x	-	x	0	0	0	x	0	0	-	-
Template Method	5	5	2	1	1	1	17	17	17	5	2	2
Visitor	1	1	0	0	0	0	2	2	2	0	0	0
State/Strategy	23	20	20	3	3	3	12	3	3	15	2	2

PI: Pattern Instances identified by approach used in this thesis CI: Common Instances

Similarly, Table 8.8 compares our results with [29] on structural design patterns using C++ baselines. The exact location of pattern instances detected by the approach was not available. As the selected systems are small examples and we manually analyzed/compared our results with the source code of these systems and validated our results.

Table 8.8: Common instances shared with [29] in C++

System	Galb++ 2.4			Libg++ 2.7.2			Mec 0.3			Socket 1.10		
Reference	[29]	[PI]	CI	[29]	[PI]	CI	[29]	[PI]	CI	[29]	[PI]	CI
Adapter	6	5	5	0	0	0	1	1	1	1	1	1
Bridge	0	0	0	1	3	1	0	0	0	0	0	0
Composite	0	0	0	0	0	0	0	0	0	0	0	0
Proxy	0	0	0	0	0	0	0	0	0	0	0	0
Decorator	0	0	0	12	12	10	0	0	0	0	0	0

PI: Pattern Instances identified by approach used in this thesis CI: Common Instances

8.5.2 Analysis of Shared Instances

The comparisons of exact shared pattern instances with complete roles give information about how two approaches differ in results. We did very deep and micro level comparisons of results with baseline approaches, which was quite intensive, laborious and time consuming task. We found wide disparity in the results of different approaches through manual analysis. For example, the approaches [19 25] extracted **53** and **5** Adapter pattern instances from JHotDraw 6.0b1, but they share "**0**" common instance. The same approaches extracted **159** and **17** instances of Adapter from Swing 1.4, but they share only "**2**" instances. Such wide disparity reflected our attention on deep analysis of results extracted by different approaches. Table 8.4 shows the common instances shared by our approach with the baseline approaches.

Table 8.9: Detailed analyses of shared instances for Adapter/Command

	[20]	[PI]
1	CH.ifa.draw.standard.AbstractFigure CH.ifa.draw.framework.FigureChangeListener	CH.ifa.draw.standard.AbstractFigure CH.ifa.draw.framework.FigureChangeListener
2	CH.ifa.draw.figures.RadiusHandle CH.ifa.draw.figures.RoundRectangleFigure	CH.ifa.draw.figures.RadiusHandle CH.ifa.draw.figures.RoundRectangleFigure
3	CH.ifa.draw.figures.GroupCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.figures.GroupCommand CH.ifa.draw.framework.DrawingView
4	CH.ifa.draw.figures.InsertImageCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.figures.InsertImageCommand CH.ifa.draw.framework.DrawingView
5	CH.ifa.draw.figures.UngroupCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.figures.UngroupCommand CH.ifa.draw.framework.DrawingView
6	CH.ifa.draw.standard.AlignCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.AlignCommand CH.ifa.draw.framework.DrawingView
7	CH.ifa.draw.standard.BringToFrontCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.BringToFrontCommand CH.ifa.draw.framework.DrawingView
8	CH.ifa.draw.standard.ChangeAttributeCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.ChangeAttributeCommand CH.ifa.draw.framework.DrawingView
9	CH.ifa.draw.standard.SendToBackCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.SendToBackCommand CH.ifa.draw.framework.DrawingView
10	CH.ifa.draw.standard.ToggleGridCommand CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.ToggleGridCommand CH.ifa.draw.framework.DrawingView
11	CH.ifa.draw.figures.TextFigure CH.ifa.draw.standard.OffsetLocator	CH.ifa.draw.figures.TextFigure CH.ifa.draw.standard.OffsetLocator
12	CH.ifa.draw.standard.AbstractTool CH.ifa.draw.framework.DrawingView	CH.ifa.draw.standard.AbstractTool CH.ifa.draw.framework.DrawingView
13	CH.ifa.draw.figures.LineConnection CH.ifa.draw.framework.Connector	CH.ifa.draw.figures.LineConnection CH.ifa.draw.framework.Connector
14	CH.ifa.draw.applet.DrawApplet CH.ifa.draw.standard.ToolButton	CH.ifa.draw.applet.DrawApplet CH.ifa.draw.standard.ToolButton
15	CH.ifa.draw.application.DrawApplication CH.ifa.draw.standard.ToolButton	CH.ifa.draw.application.DrawApplication CH.ifa.draw.standard.ToolButton
16	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.Drawing	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.Drawing
17	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.DrawingEditor	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.DrawingEditor
18	CH.ifa.draw.contrib.PolygonHandle CH.ifa.draw.framework.Locator	CH.ifa.draw.contrib.PolygonHandle CH.ifa.draw.framework.Locator
19	CH.ifa.draw.standard.LocatorHandle CH.ifa.draw.framework.Locator	CH.ifa.draw.standard.LocatorHandle CH.ifa.draw.framework.Locator
20	CH.ifa.draw.standard.ToolButton CH.ifa.draw.util.PaletteIcon	CH.ifa.draw.standard.ToolButton CH.ifa.draw.util.PaletteIcon
21	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.Painter	CH.ifa.draw.standard.StandardDrawingView CH.ifa.draw.framework.Painter
22	CH.ifa.draw.standard.ReverseFigureEnumerator CH.ifa.draw.util.ReverseVectorEnumerator	CH.ifa.draw.standard.ReverseFigureEnumerator CH.ifa.draw.util.ReverseVectorEnumerator

Several approaches extract the same numbers of patterns, but the worst cases are noticed when extracted instances are completely different. It is also noticed that some

approaches share partial role of patterns, but they miss the complete roles. We did the micro analysis of comparing common instances with baselines. Table 8.9 illustrates the instances of Adapter/Command design pattern extracted by strong baseline [20] and shared with our approach. We cannot display complete roles of common instances due to limitation of space. The complete list of our extracted results will be available on web for comparisons. It was really surprising that approach [45] has detected **28** instances of Adapter from JHotDraw 5.1, but only "**1**" instance is true. Similar examples are analyzed in results of different other approaches as well. Such approaches have major focus on improving recall, but they compromise on very low precision.

Our approach has detected instances of Adapter, which are not shared with the approach [20] as shown in Table 8.10. The manual inspection of the source code realizes the existence of these patterns according to GoF pattern specifications. These instances are missed by the approach [20] due to different pattern interpretations or loose criteria used for detection of patterns.

Table 8.10: Non-Shared instances in JHotDraw 5.1 with [PI]

	Adapter	Adaptee
1	CH.ifa.draw.Contrib.PolygonTool	CH.ifa.draw.Contrib.PolygonFigure
2	CH.ifa.draw.Standard.ConnectionTool	CH.ifa.draw.Standard.ConnectionFigure
3	CH.ifa.draw.Standard.CreationTool	CH.ifa.draw.Framework.Figure
4	CH.ifa.draw.Standard.HandleTracker	CH.ifa.draw.Framework.Handle
5	CH.ifa.draw.Standard.Commandattributefigure	CH.ifa.draw.Framework.Figure
6	CH.ifa.draw.figures.AttributeFigure	CH.ifa.draw.figures.FigureAttributes

8.6 Discussion

This section discusses the extracted results, the precision and recall of results and the overall contribution of approach to meet our desired goals.

The approach has extracted different implementation variants of design patterns instances from source code of multiple languages (Java, C# and C/C++) with significant improvement in precision and recall. Initially, we tested our approach on single pattern definitions (source code examples) as proof of concept for the validation of approach. In the continuation, Junit 3.7 was the interesting example for us, because it contains only 52 classes and interfaces and it is implemented by using different design patterns. It was relatively easy to validate our results manually using this example from the source code. The 98% precision and 100% recall from Junit 3.7 motivated us to test our approach on other large size open source systems to ensure the scalability of the presented approach. Further, experiments are performed on the examples, which are used by the baselines approaches. The extracted precision and recall is shown in Table 8.4. We calculated the combined effect of precision and recall as F-score shown in Figure 8.3. The F-score for JUnit 3.7, JHhotDraw 5.1, JRefactory 2.6.24 and QuickUML 2001 are (99%, 97%, 97%

and 71%) respectively. Our detected precision and recall are same or better in some cases when some discrepancy is found in the claimed results of baseline approaches. For Apache Ant 1.6.2 system, we selected weak baselines [19 25] for comparisons. The precision is compared with [25], because they performed experiments on more patterns and results of approach are available for validation. We did not calculate the recall for Apache Ant 1.6.2, because the weak baselines do not mention recall values. However, precision for Apache Ant 1.6.2 is 68%. The deep analysis of common instances shows convergence and divergence of our extracted results with the baseline approaches. The approach used in this thesis focused on Java examples in particular, but it can be generalized for other languages. The precision and the recall for C++ examples are same as extracted by [29], which validate our extracted results. The examples and baseline results for C# were not available and we tested our pattern definitions on GoF source code examples from [123]. The overall goals of approach are achieved as set requirements for approach in Section 3.5.

I) The approach has detected structural, implementation and elemental instances of design patterns with the help of customizable pattern definitions.
II) The approach extracted patterns from source code of multiple languages including Java, C#, C++, but it can be extended for languages supported by Enterprise Architect Modeling Tool.
III) We have achieved improved precision and recall rates on diverse examples as indicated in Section 8.4.
IV) We analyzed and detected disparity in the results of different approaches and communicated our findings to the authors of different approaches. This can leads to the accuracy of baselines results.

8.7 Threats to Validity

Validity is the key challenge for researchers and practioners in conducting empirical research work. Validity is defined as: “the best available approximation to the truth of a given proposition, inference, or conclusion”. Reference [120] states that for empirical research to be acceptable as a contribution to scientific knowledge, the researcher needs to convince related academia and industry that conclusions drawn from an empirical study are valid. This section discusses threats for the validation of our results.

A major threat to the results of our approach is the lack of standard definitions for design patterns and unavailability of trustable benchmarks for validation of our results. While one can precisely define variants of a pattern, but there is no agreed upon definition of different variants for each design pattern to-date. We took results of approach [20] as strong baseline to compare our results. However, some discrepancies are noticed in the

results of this approach. The results of our approach are very close to the results of baseline approaches in some patterns and with improved precision and recall in the case of other patterns. The approaches [45 66] are taken as moderate baselines, because detailed results of these approaches are available on web [126] for comparisons. The other approaches such as [19 25] are taken as weak baselines. So our results against these approaches have threats to their validity. The threats to experimental validity can be classified as:

Internal Validity

Internal validity is concerned with the consistency of the measurements, appropriate use of tools, and methods [127]. The prototyping tool validates our approach on different software systems with different sizes. Internal validity is affected by the experimental bias. We tried to manually analyze our extracted results to eliminate the number of false positives, but the numbers of false negatives are not analyzed manually which may affect the accuracy of approach. The community should have access to the experimental results to eliminate effect of biasness. All the experimental results of our approach will be available on the web and researchers can validate our results.

External Validity

External validity threats concern the generalization of results. We selected five examples of Java programs ranging from small to large size and all GoF patterns with different variants, but we cannot generalize precision and recall for all design patterns. The precision and recall can vary in the case of other languages and large size legacy systems. Although, feature types including SQL queries and regular expressions are general and can be used to extract patterns from the source code of other languages, but the source code parser module for each new language requires additional effort to generalize approach for all languages. The user defined features types can be extended to generalize the results for all object oriented languages and design patterns.

Construct Validity

Construct validity threats involve the relation between theory and observation. There is also still no consensus on the possible number of variants for each design pattern. It is possible that our feature types and pattern definitions do not take into consideration all the possible variants of each design pattern due to new implementation features of programming languages, which can affect the accuracy of presented approach. However, customizable pattern definitions allow user to customize pattern definitions for recognizing false negatives.

Reliability Validity

The extent to which results are consistent over time and an accurate representation of the total population under study is referred to as reliability and if the results of a study can be reproduced under a similar methodology, then the research instrument is considered to be reliable [113]. This reliability affects the replicability of our results. The selected examples are all open source systems and source code is available on web for validation. Reliability validity threats will be eliminated because our prototyping tool, experimental data and results will be available on the web.

8.8 Summary

This chapter discussed the evaluation of presented approach based on extracted pattern instances, precision, recall, F-score and disparity in the results of different approaches. The assumptions which are taken during the pattern detection process are clearly stated in Section 8.1. Section 8.2 presented sample experimental setup used for the evaluation of approach. The justifications for selection of experimental examples are clearly stated. The extracted results from examined setup examples are presented in Section 8.3. Section 8.4 discussed the precision, the recall and the F-Score of extracted results in comparison with baselines. The disparity of results rendered by different approaches is analyzed and discussed in Section 8.5. The common/shared instances of patterns in comparison with baseline approaches are highlighted in this section. Section 8.6 discussed results, precision, recall, F-score and overall worth of approach to meet our required objectives. Section 8.7 discussed the implications to validity of our results.

Chapter 9

Conclusions and Future Work

This chapter discusses the conclusions, the lessons learned and the possible future research directions from this thesis. The detection of design patterns from different legacy applications support reverse engineering, program comprehension, refactoring, restructuring, maintenance and reengineering disciplines. The numbers of approaches have been proposed and implemented, but they still lack accuracy, flexibility, customization and effectiveness. The discoveries of new patterns for the development of applications pose challenges for existing and new pattern recovery approaches to cope with new patterns. The approaches which used hard coded algorithms for the detection of patterns are not flexible for detecting structural and implementation variants of different design patterns. The approaches using pure formal techniques for specification of patterns hinder user for customizing pattern specifications to detect the implementation variants. Similarly, the approaches which extract only few relatively easy patterns on small examples cannot be generalized.

Section 9.1 discusses concepts of the approach and summarizes end results that realize concepts. It concludes overall efforts and contributions of the thesis. The critical review of the approach is presented and summarized in Section 9.2. Finally, Section 9.3 discusses future research directions and open issues in the area of design pattern recovery.

9.1 Conclusions and Summary

The goal of approach used in this thesis was to recognize design patterns and their variants from source code of multiple languages accurately. In order to achieve this goal, we reviewed different pattern recovery approaches and tools used in recent decade for detection of design patterns. Chapter 2 discussed the problem of variants, which hampers pattern recovery approaches and tools. The variations in design pattern definitions became requirement for differentiating and detecting various variants. The concept of customizable pattern definitions is used to address this requirement. Chapter 3 discussed in detail related work of different approaches and highlighted major problems which further flourished motivation for this thesis. An up-to-date overview of latest and successful recovery approaches is presented in Table 3.1. We concluded in review that most of pattern recovery approaches used single recognition technique for detecting

design patterns. We conceived the goal of multiple pattern recognition techniques and multiple language support for pattern detection from Section 3.1. We experienced that accuracy of design pattern detection tools depend on the interpretation of a design pattern and its variants. We discussed and compared the features of different tools in Table 3.3. The evaluation of different tools motivated us to develop our custom build prototyping tool, which is used for the realization of presented approach. Further, the analysis of wide disparity in results of different approaches fostered for deep and intensive analysis of diverse results. We spent countless hours to explore the causes of disparity in the results of different approaches. Chapter 3 concludes with observations, lessons learned and clear requirements, which became motivation for the presented approach. Chapter 4 discussed main concept of the approach, which is based on the requirements elaborated in previous chapter. The concept of variant pattern definition creation process is introduced to define and recognize the structural as well as the implementation variants of design patterns. The goal is realized by using customizable and reusable feature types. In order to improve the accuracy of pattern recognition process, we used concept of multiple searching techniques discussed in Section 4.3. Finally, the chapter concludes with major challenges of approach and the concepts used to handle challenges. Chapter 5 discussed in detail the concept of first phase of Chapter 4 with major focus on pattern definition creation process. The feature types with different arguments are backbone for pattern definitions. Examples of different design pattern definitions with variants based on feature types address (requirement 1, 2 and challenge 1). The pattern recognition approach based on multiple techniques is discussed in Chapter 6. The objectives and scope of pattern recognition process are clearly described. The concept of SQL queries, regular expressions, source code parser module and annotations realized the power of approach for pattern detection. The translation of feature types into SQL/REGX/PMF dominated our approach for pattern detection. Finally, the static and the dynamic views of pattern recognition process are discussed at end. The prototyping tool realized the concept of approach and has successfully recovered patterns from source code of different applications. The unique features of prototyping tool are discussed in Section 7.4. The results from the approach and prototyping tool are presented in Chapter 8. Experiments are performed on GoF source code examples of Java [121], C# [122], C/C++ [123] to validate the pattern definitions used for detection of patterns. Furthermore, experiments are performed on different software systems ranging in size from 43-883 classes. Precision, recall and F-score metrics are calculated and presented in this section. Disparity in results extracted by different tools is analyzed and our results are compared with other baseline approaches to remove the disparity through manual analysis of results. The removal of disparity in the results of different approaches can leads to trusted benchmarks, which can be used by the groups working on benchmarking for design pattern recovery. The validity of approach and the scope is discussed at the end in Section 8.7.

9.2 Critical Review

Assumptions

We accepted results of approaches [19 20 25 45 66] as baselines to compare our results with these approaches. We found some discrepancies in the results of these approaches through manual analysis of source code. One major cause of disparity is different interpretation of pattern definitions by different approaches. The results for patterns that have no baselines remain threat to validation and accuracy of approach. We analyzed extracted patterns instances manually, but we did not check the whole source code manually to analyze false negatives.

Use of Annotations as Manual Effort

The approach suggests/requests developers to use annotations related with design patterns, which are helpful for pattern extraction and documentation. Currently, we annotated partially the source code of some applications to prove the efficacy of annotations. The source code is manually annotated which requires a lot of manual effort. However, tool support is available to automate annotations in the source code as mentioned in [110]. Finally, we used annotations for patterns which cannot be extracted with static and dynamic information.

Limitation of Source Code Parser Module

The source code parser module is currently able to support Java and C# source code for extracting artifacts from the source code. Different parsers such as (delegation parser, aggregation parser, method calling parser, etc.) are implemented to support structured analysis and improve accuracy of pattern detection. The parser module is based on grammar of CoCo/R parser grammar. The limitation in grammar of CoCo/R parser for all versions of Java and C# is also obstacle to support latest version of these languages. The applicability of parser module to other languages also requires additional effort. It is possible to develop parsers for different languages that can be integrated with our prototype. The development of parser modules for different languages and their integration is the flexibility of our approach, but on the other hand it is quite laborious, intensive and time consuming.

Integration with IDEs

The prototyping tool is developed as Add-In with EA tool which have further provision for integration with different other environments. It completely supports the .Net Framework. The integration of input/out formats of our tool with the other tools needs

further investigation. The output of our tool for visualization of design patterns is obvious advantage.

Scope and Validity Threats

The approach is scalable because we performed experiments on systems with different sizes ranging from small to large scale. However, we did not perform experiments on industrial applications. The approach has generalized framework and it can be extended for different patterns and other languages. The performance was not the main concern for this thesis. The validity threats are still open question for all design pattern detection approaches and we mentioned certain assumption of our approach in Section 8.1.

Disparity of Results

The disparity of results by different approach is still debatable and there is no common solution for this problem. We used customizable pattern definitions to reduce the number of false positives and false negatives.

9.3 Future Work

The possible future research directions of approach used in this thesis are the following:

- The major disappointment for design pattern recovery approaches is the disparity of results rendered by different approaches from same systems. Some approaches perform experiments on very small examples and show 100% precision and recall, but when these approaches are tested on a large system, the claimed accuracy falls down. We suggest the following guidelines for solution of this problem as future work.

I) Trusted Benchmark systems/ baselines to compare and cross validate the results of different approaches.

II) Definitions of possible structural as well as implementation variants for each design pattern.

- It is possible to enhance the capability of prototyping tool for detecting patterns beyond the GoF patterns. The focus of approach was to allow user for customizing pattern definitions. Pattern definitions can be extended to detect J2EE and architectural patterns.
- Most legacy applications are developed using Java, C#, C/C++ languages. Parser module is currently supporting Java and C#. The parser module for C/C++ can be developed and integrated using same grammar from CoCo/R parser.
- The input and out formats used by design pattern recovery tools are very important for integration of different tools. Output of presented prototyping tool can be used by different modeling tools for comprehension and visualization.

- Composition of design patterns is very important because different design patterns are used for plumbing the frameworks which are called architecture patterns. The architecture patterns are composed from different design patterns. Our approach can be extended to recover the architecture patterns which ultimately help to recover the design model of legacy applications.
- Visualization of extracted patterns results is another future extension of our tool, because it plays key role for the comprehension of extracted patterns results. The XMI structure generated by EA Modeling Tool can be used for presentation of results.
- It is possible to extend approach for automatic translation of feature types into SQL queries and regular expressions.
- The approach can be extended by automatic translating UML structure of each design pattern into feature types used for the specifications of design patterns.

List of Figures

List of Tables

Bibliography

[1] Jing Dong, Yajing Zhao and Tu Peng, "A Review of Design Pattern Mining Techniques", The International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Volume 19, Issue 6, pp. 828-855, September 2009, ISSN: 1793-6403, DOI: 10.1142/S021819400900443X.

[2] Dirk Beyer and Claus Lewerentz, "CrocoPat: efficient pattern analysis in object-oriented programs", In Proceedings of the International Workshop on Program Comprehension (IWPC'03), Portland, OR, USA, pp. 294–295, 2003, ISBN: 0-7695-1883-4.

[3] Ilka Philippow, Detlef Streitferdt, Matthias Riebisch and Sebastian Naumann, "An approach for reverse engineering of Design patterns", Journal of Software and System Modeling, Volume 4, No. 1, pp. 55-70, Springer-Verlag 2004, ISSN: 1619-1374, DOI: 10.1007/s10270-004-0059-9.

[4] Capers Jones, "Geriatric Issues of Aging Software", In Journal of Defense Software Engineering, pp. 4-8, Volume 20, No. 12, December 2007, ISSN: d0000089.

[5] Niklas Pettersson, Welf Löwe and Joakim Nivre, "On Evaluation of Accuracy in Pattern Detection", First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06), pp. 1-8, October 2006, ISBN: 0-7695-2719-1.

[6] Jing Dong and Yajing Zhao, "Classification of Design Pattern Traits", In Proceedings of Nineteenth International conference on software engineering and knowledge (SEKE), pp. 473-476, Bostan, USA, July 2007, ISBN: 1-891706-20-9.

[7] Marek Vokac, " An efficient tool for recovering design patterns from C++ code", Journal of Object Technology, Volume 5, No. 1, pp. 139–157, Jan-Feb 2006, ISSN: 1660-1769.

[8] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom and Welf Lowe, "Automatic design pattern detection", In Proceedings of 11th International Workshop on Program Comprehension (IWPC), USA, pp. 94-103, May 2003, ISBN: 0-7695-1883-4.

[9] Rainer Koschke, "What architects should know about reverse engineering and reengineering", In Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 6-10, Pittsburgh, Pennsylvania ,November 2005, ISBN: 0-7695-25482.

[10] Nija Shi, "Reverse Engineering of Design Patterns from Java Source Code", PhD thesis, University of California, USA, pp. 2, 2007.

[11] Lothar Wendehals, "Improving design pattern instance recognition by dynamic analysis" , In Proceedings of the ICSE workshop on Dynamic Analysis (WODA), pp. 29-32, Portland, Oregon, May 2003, ISBN: 0-7695-1877-X.

[12] Jason M. Smith and David Stotts "SPQR: Flexible automated design pattern extraction from source code", In Proceedings of Automated Software Engineering, pp. 215-224, IEEE Computer Society Press, Canada, October 2003, ISBN: 0-7695-2035-9.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software", Addison-Wesley Publishing Company, Reading, MA, 1995, ISBN: 0201633612.

[14] Hausi A. Muller, Kenny Wong and Scott A. Tilley, "Understanding Software Systems Using Reverse Engineering Technology", In Proceedings of 62nd Congress of "L'Association Canadiene Francaise pour l'Avancement des Sciences (AFCAS)", Montreal Canada, May 16-17, 1996.

[15] IBM Executive Brief, Legacy Transformation: Finding New Business Value in Older Applications, 2005, http://www306.ibm.com/software/info/features/bestofboth.

[16] Gregory Tassey, "The Economic Impact of inadequate Infrastructure for software testing", National Institute of Standards and Technology, Final Report by (RTI), 2002, http://www.nist.gov/director/planning/upload/report02-3.pdf.

[17] Nucleus: Nucleus Research Report: Microsoft Patterns and Practices, August 2009 http://msdn.microsoft.com/en-us/practices/ee406167.aspx.

[18] Jikes Source: http://jikes.sourceforge.net/ [accessed 02-10-2009].

[19] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino and Michele Risi, "Design pattern recovery through visual language parsing and source code analysis", The Journal of Systems and Software, Volume 82, Issue 7, pp. 1177–1193, 2009, ISSN: 0164-1212.

[20] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides and Spyros T. Halkidis, "Design Pattern Detection Using Similarity Scoring", IEEE Transactions on Software Engineering, Volume 32, No. 11, pp. 896-909, November 2006, ISSN: 0098-5589.

[21] James Noble, "Towards a Pattern Language for Object Oriented Design", In Proceedings of Technology of Object-Oriented Languages and Systems, pp. 2-12, Melbourne, Australia, November 1998, ISBN: 0-7695-0053-6.

[22] Scientific Toolworks Inc. Understand for C++, 2003, http://www.scitools.com/.

[23] Krzysztof Stencel and Patrycja Wegrzynowicz, "Detection of Diverse Design Pattern Variants", In Proceedings of 15th Asia-Pacific Software Engineering Conference, pp. 25-32, Beijing, China ,3-5 December 2008, ISBN: 978-0-7695-3446-6.

[24] Vojislav D. Radonjic and Jean-Pierre Corriveau, "Making Patterns Better Design Tools: Requirements Analysis for a Family of Navigators for Design Pattern Catalogs, In Proceeding of IASTED Conference on Software Engineering, pp. 349-354, Innsbruck, Austria, February 15-17, 2005, ISBN: 0-88986-466-7.

[25] Nija Shi and Ronald A. Olsoon "Reverse Engineering of Design Patterns from Java Source Code", In Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Volume 00, pp. 123-134, 2006, ISBN: 0-7695-2579-2, ISSN: 1527-1366.

[26] Sparx System Architect Modeling tool. <http://www.sparxsystems.com/products/ea/> [accessed 05.05.09].

[27] Christian Kramer and Lutz Prechelt, "Design recovery by automated search for structural design patterns in object oriented software", In Proceedings of Working Conference on Reverse Engineering (WCRE'96), Monterey, CA, USA, pp. 208–215, ISBN: 0-8186-7674-4.

[28] Zsolt Balanyi and Rudolf Ferenc, "Mining design patterns from C++ source code", In Proceedings of International Conference on Software Maintenance (ICSM'03), pp. 305-314, Amsterdam, The Netherlands, 2003, ISBN: 0-7695-1905-9.

[29] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi, "Case studies of visual language based design patterns recovery", In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR), pp. 165-174, Bari, Italy, March, 2006, ISBN: 0-7695-2536-9, ISSN: 1052-8725.

[30] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino and Michele Risi, "A Two Phase Approach to Design Pattern Recovery", In Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR'07), pp. 297-306, Amsterdam, Netherlands, 21-23 March 2007, ISBN: 0-7695-2802-3.

[31] Hironori Washizaki, Kazuhiro Fukaya, Atsuto Kubo and Yoshiaki Fukazawa, "Detecting Design Patterns Using Source Code of Before Applying Design Patterns", In Proceedings of Eight IEEE/ACIS International Conference on Computer and Information Science", pp. 933-938, Shanghai, China, June 2009, ISBN: 978-0-7695-3641-5.

[32] J.C. Silva, Jose Creissac Campos, and João J. Saraiva, "Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications", Springer-Verlag Berlin Heidelberg, Volume 4323, pp. 137-150, May 2007, ISBN:978-3-540-69553-0.

[33] Damir Kirasic and Danko Bash, "Knowledge-Based Intelligent Information and Engineering Systems", Lecture Notes in Computer Science, Volume 5177/2008, September 20, 2008, ISBN: 978-3-540-46542-3.

[34] Jason M. Smith and David Stotts, "Elemental Design Patterns: Case Studies in Automated Design Pattern Detection in C++ Code using SPQR", Technical Report TR05-013, University of North Carolina, 2004.

[35] Jing Dong, Dushyant S. Lad and Yajing Zhao, "DP-Miner: Design Pattern Discovery Using Matrix." In Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS), pp. 371-380, Arizona, USA, March 2007, ISBN: 0-7695-2772.

[36] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele, "Design pattern mining enhanced by machine learning", In Proceedings of the 21st International Conference on Software Maintenance, pp. 295-304, Budapest, Hungary, September, 2005, ISBN: 0-7695-2368-4, ISSN: 1063-6773.

[37] Hong Zhu, Ian Bayley, Lijun Shan and Richard Amphlett, "Tool Support for Design Pattern Recognition at Model Level", In Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference, pp. 228-233, Seattle, Washington, USA , July 20-24, 2009, ISSN: 0730-3157.

[38] Jing Dong, Yongtao Sun and Yajing Zhao, "Compound record clustering algorithm for design pattern detection by decision tree learning", In Proceedings of Information Reuse and Integration, pp. 226-231, Las Vegas, Nevada, USA ,13-15 July 2008, ISBN: 978-1-4244-2659-1.

[39] Yann-Gael Gueheneuc, Houari Sahraoui, and Farouk Zaidi, "Fingerprinting design patterns", In Proceedings of the 11th Working Conference on Reverse Engineering(WCRE), pp. 172-181, Victoria, BC. Canada, November, 2004, ISBN: 0-7695-2243-2, ISSN: 1095-1350.

[40] Marcel Birkner, "Object-Oriented Design Pattern Detection using Static and Dynamic Analysis In Java Software", Master Thesis, University of Applied Sciences Bonn, Germany , pp. 95-98, August 2007.

[41] Jing Dong and Khang Zhang, "Visualizing design patterns in their applications and applications", IEEE Transactions in Software Engineering, Volume 33, No. 7, pp.433-453, July 2007, ISSN: 0098-5589.

[42] Kamran Sartipi and Lei Hu ,"Behavior-Driven Design Pattern Recovery", IASTED International Conference on Software Engineering and Applications (SEA 2008), pp. 179-185, Orlando Florida USA, November 16-18 2008, ISBN: 978-0-88986-776-5.

[43] Matthias Meyer and Lothar Wendehals, "Selective tracing for dynamic analyses", In Proceedings of the 1st Workshop on Program Comprehension through Dynamic Analysis, pp.33–37, Pittsburgh, PA, USA, 2005,

http://www.lothar-wendehals.de/publikationen/dokumente/MW05.pdf.

[44] Eclipse Foundation: The Eclipse Platform, Online at http://www.eclipse.org, [Accessed on January 2010].

[45] Yann-Gael Gueheneuc and Giuliano Antoniol, "DeMIMA: A Multilayered Approach for Design Pattern Identification", IEEE Transactions on Software Engineering, Volume 34, No. 5, pp. 667-684, 2008, ISBN: 0098-5589.

[46] Francesca Arcelli and Luca Cristina, "Enhancing Software Evolution through Design Pattern Detection", In Proceedings of the 3rd International Workshop on Software Evolvability (PCODA'08), pp. 7-14, Paris, France ,October 2007 , ISBN: 0-7695-3002-8/07.

[47] Francesca Arcelli, Fabrizio Perin, Claudia Raibulet and Stefano Ravani, "Behavioral Design Pattern Detection through Dynamic Analysis", In Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA'08), pp. 11-16, Antwerp, Belgium ,October 2008.

[48] Hakjin Lee, Hyunsang Youn and Eunseok Lee, "Automatic Detection of Design Pattern for Reverse Engineering", In Proceedings of 5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007), pp.577-583, 2007, ISBN:0-7695-2867-8.

[49] Adrian Paschke, "A Semantic Design Pattern Language for Complex Event Processing", Association of Advancement in Artificial Intelligence (www.aaai.org), pp. 54-60, 2009, http://www.aaai.org/Library/Symposia/Spring/2009/ss09-05-010.php.

[50] Awny Alnusair and Tian Zhao, "Towards a Model-driven Approach for Reverse Engineering Design Patterns", In Proceedings of the 2nd International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE 2009), pp. 1-15, Denver, Colorado, USA, October 4, 2009, http://www.uni-koblenz.de/confsec/twomde2009/paper07.pdf.

[51] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur and Yarden Kartz, "A Practical OWL-DL Reasoner", In Journal of Web Semantics: Science,

Services and agents on the World Wide Web, pp. 51-53, Volume 5, Issue 2, 2007, ISSN: 1570-8268.

[52] Yingxu Wang and Jian Huang, "Formal Modeling and Specification of Design Patterns Using RtPA", International Journal of Cognitive Informatics and Natural Intelligence, Volume 2, Issue 1, pp. 100-111, 2008, ISSN: 1557-3958.

[53] Ian Bayley and Hong Zhu , "On the Composition of Design Patterns", In Proceedings of Eighth International Conference on Quality Software, pp. 27-36 Oxford, UK,12–13 August 2008, ISBN:978-0-7695-3312-4, ISSN: 1550-6002.

[54] Ian Bayley and Hong Zhu, "Formalizing Design Patterns in Predicate Logic", In Proceedings of Fifth IEEE International Conference on Software Engineering and Formal Methods, pp. 25-34, London, England, September 2007, ISBN:0-7695-2884-8.

[55] Ian Bayley and Hong Zhu, "Formal specification of the variants and behavioral features of design patterns", The Journal of systems and Software, Volume 83, Issue 2, pp.209-221, October 2009, ISSN: 0164-1212.

[56] Amnon H. Eden, Jonathan Nicholson and Epameinondas Gasparis, "Formal specification of design patterns in LePUS3 and Class-Z", Technical Report CSM-472, ISSN: 1744-8050, December 2007.

[57] Gennaro Costagliola, Andrea De Lucia, R. Francese, Michele Risi and Giuseppe Scannello, "A Component-Based Visual Environment Development Process", In Proceedings of International Conference on Software Engineering and Knowledge Engineering (SEKE'02), pp. 327- 334, Ischia, Italy, 2002, ISBN:1-58113-556-4.

[58] Homepage of FrontEndART Software Ltd, http://www.frontendart.com.

[59] Jing Dong and Yajing Zhao, "Experiments on Design Pattern Discovery", In Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering (PROMISE), in conjunction with ICSE, Minneapolis, MN, USA, May 2007, ISBN:0-7695-2954-2.

[60] James M. Bieman, Greg Straw, Huxia Wang ,P. Willard Munger and Roger T. Alexander, "Design patterns and change proneness: an examination of five evolving systems", Software Metrics Symposium, In Proceedings of Ninth International Symposium on Software patterns, pp. 40–49, Sydney, Australia, 3-5 Sept. 2003, ISBN:0-7695-1987-3.

[61] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta and Roberto Fiutem" ,Object-oriented design patterns recovery ", Journal of Systems and Software, Volume 59, No. 2, pp. 181-196, November 2001, ISSN: 0164-1212.

[62] Galb++ source code: http://lancet.mit.edu/ga/ [accessed 07.02.10].

[63] Libg++ source code: ftp://ftp.gnu.org/gnu/libg++/ [accessed 07.02.10].

[64] Olivier Kaczor, Yann-Gael Gueheneuc and Sylvie Hamel, "Efficient Identification of Design Patterns with Bit-vector Algorithm", In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR), pp. 175-184, Bari Italy, March 2006, ISBN: 0-7695-2536-9, ISSN: 1052-8725.

[65] Wei Wang and Vassilios Tzerpos, "Design pattern detection in Eiffel systems" In Proceedings of 12th Working Conference on Reverse Engineering (WCRE), pp. 1-10, Pittsburgh, Pennsylvania 7-11 November 2005, ISBN: ISBN: 0-7695-2474-5.

[66] Yann-Gael Gueheneuc, Jean-Yves Guyomarc'h and Houari Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study", Software Quality Journal, Volume 18, Issue 1, pp. 145-174, 2010, ISSN: 0963-9314 , DOI :10.1007/s11219-009-9082-y.

[67] Niklas Pettersson, Welf Löwe and Joakim Nivre, "Evaluation of Accuracy in Design Pattern Occurrence Detection", IEEE Transactions on Software Engineering, IEEE computer Society Digital Library, IEEE Computer Society, pp. 575-590, Volume 36, No. 4, July/August 2010, ISBN: 0098-5589.

[68] Marjan Hericko and Simon Beloglavec, "A composite design-pattern identification technique", Informatica 29, pp 469–476, 2005, ISSN: 0350-5596.

[69] Dania Harb, Cédric Bouhours and Hervé Leblanc, "Using an Ontology to Suggest Software Design Patterns Integration", Lecture Notes in Computer Science, Volume 5421/2009, pp. 318-331, April 28, 2009, ISBN: 978-3-642-01647-9.

[70] Narain Gehani, "Specifications: Formal and informal - a case study", Software Practice and Experience, Volume 12, Issue 5, pp. 433 – 444, 27 October 2006, ISSN:00380644.

[71] Pawan Vora, "Web Application Design Patterns", Morgan Kaufmann Publications, 2009, ISBN: 978-0-12-374265-0.

[72] Dirk Beyer and Andreas Noack, "CrocoPat 2.1, Introduction and Reference Manual", Report No UCB//CSD-04-1338, Computer Science Division (EECS), University of California Berkeley, California. 2004.

[73] Toufik Tabi, David Chek and Ling Ng, "Modeling of Distributed Objects Computing Design Pattern Combinations using Formal Specification Language", International Journal of Applied Mathematics and Computer Science, Volume 13, No. 2, pp. 239-253, 2003, ISSN: 1641-876X.

[74] Christopher P. Fuhrman, "Lightweight models for interpreting informal specifications", Requirements Engineering Journal, Volume 8, No. 4 pp. 206–221,November, 2003, ISSN: 1432-010X, DOI 10.1007/s00766-002-0154-9.

[75] Daniel Petri and Gyorgy Csertan, "Design Pattern Matching", Periodica Polytechnica SER. EL. ENG. Volume 47, No. 3–4, pp. 205–212, 2003, ISSN 0302-9743.

[76] Gerson Sunye, Alain Le Guennec and Jean-Marc Jezequel, "Design Pattern Application in UML", In Proceedings of ECOOP'00, LNCS 1850, pp. 44-62, ,http://www.ifs.unilinz.ac.at/~ecoop/cd/papers/1850/18500044.pdf.

[77] Jean-Marc Rosengard and Marian F. Ursu, "Ontological representations of software patterns", Lecture Notes in Computer Science (Proceedings of the KES'04), Volume 3215, pp. 164-179, 2004.

[78] Dey-Kyoo Kim, Robert B. France, Sudipto Ghosh and Enjee Song "A UML-Based Language for Specifying Domain-Specific Patterns", Journal of Visual Languages and Computing, Special Issue on Domain Modeling With Visual Languages, Volume 15, No. (3-4), pp. 265-289, June - August 2004, ISSN: 1045-926X.

[79] Silvio Meier , Tobias Reinhard, Reinhard Stoiber and Martin Glinz, "Modeling and Evolving Crosscutting Concerns in ADORA", In Proceedings of the 11th Workshop on Early Aspects at the International Conference on Software Engineering, pp. 91-Minneapolis, USA, 2007, ISBN:0-7695-2830-9.

[80] Luca Sabatucci, Massimo Cossentino, and Angelo Susi, "Introducing Motivations in Design Pattern Representation", Lecture Note in computer Science, Volume 5791, pp. 201-210, September 2009, ISSN: 1611-3349, DOI: 10.1007/978-3-642-04211-9.

[81] Martin Gogolla and Mark Richters, "On combining semi-formal and formal object specification techniques", Lecture Notes in Computer Science, Volume 1376, pp. 238-252, Springer Berlin, Heidelberg, April 2006, ISSN: 1611-3349. DOI: 10.1007/3-540-64299-4.

[82] Deni Torres-Roman, Joaquin Cortez-Gonzlaez, Raul Ernesto and Gonzalez-Torres, "Improving the Digital Design with Semi-formal Specification", In Proceedings of 16th International Conference on Electronics, Communications and Computers (CONIELECOMP'06), pp. 55-55, Cholula, Puebla, Mexico, February 27-March 01, 2006, ISBN: 0-7695-2505-9.

[83] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun, "Precise Modeling of Design Patterns in UML", In Proceedings of 26th International Conference on Software Engineering, pp.252-261, Edinburgh, Scotland, United Kingdom, May 2004, ISSN: 0270-5257.

[84] Zhi-Xiang Zhang, Qing-Hua Li' and Ke-Rong Bem, "A new Method for design pattern mining ", In Proceedings of the Third International Conference on Machine Laming and Cybernetics, pp. 1755-1759, Shanghai, China, August 26-29, 2004, ISBN: 0-7803-8403-2.

[85] Martin Flower, "Patterns of Enterprise Application Architecture", Addison-Wesley Professional, 2002, ISBN: 0321127420.

[86] David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk and David Lavigne, "Enterprise Solution Patterns using Microsoft.NET Version 2.0, Microsoft Corporation, 2003. <http://www.willydev.net/descargas/PartnerAndPractices/WillyDev_ESP.pdf>.

[87] Imed Hammouda and Jakub Rudzki, "Classification of Design Patterns", A Course Report, Tampere University of Technology, 2004. http://www.cs.tut.fi/~kk/webstuff/PatternClassification.pdf.

[88] Marek Voka, "On the practical use of software design patterns" , PhD Thesis, pp. 18-20, University of Oslo, Norway.

[89] Thomas Erl, "SOA Design Patterns", Prentice Hall Publications, pp. 100-101, 2008, ISBN: 0-13-613516-1.

[90] Christopher Thomson and Mike Holcombe," Using a formal method to model software design in XP projects", In Journal of Annals of Mathematics, Computing & Teleinformatics, Vol 1, No 3, pp. 44-53, 2005, ISSN: 1109-9305.

[91] Judith Bishop, "C# 3.0 Design Patterns", O'Reilly Media, Inc, 2007, ISBN: 0-596-52773-x.

[92] Mark Grand, "Java Enterprise Design Patterns: Patterns in Java Volume 3", Wiley, 2001, ISBN: 978-0471333159.

[93] Steve Holzner, "Design Patterns for Dummies", Dummies Publishers, 2006, ISBN: 13:978-0471798545.

[94] Kent Beck , James O. Coplien , Ron Crocker , Lutz Dominick , Gerard Meszaros, Frances Paulisch, and Siemens Ag," Industrial Experience with Design Patterns", In Proceedings of 18th Intl. Conference on Software Engineering, pp.103-114,1996.

[95] Frank Buschmann, Kevlin Henney and Douglas C. Schmidt, "Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages", Wiley, 2007, ISBN: 978-0471486480.

[96] Krzysztof Stencel and Patrycja Wegrzynowicz, "Implementation Variants of the Singleton Design Pattern", Lecture Notes in Computer Science, Volume 5333, pp.

396-406, November 19, 2008, ISSN: 1611-3349, DOI: 10.1007/978-3-540-88875-8_61.

[97] Fujaba Home Page :< http://wwwcs.uni-paderborn.de/cs/fujaba/> [accessed 8.06.10].

[98] PINOT Home Page: http://www.cs.ucdavis.edu/~shini/research/pinot/ [accessed 8.06.10].

[99] Karthik Soundararajan and Robert W. Brennan, "A Proxy Design Pattern to Support Real-Time Distributed Control System Benchmarking", Lecture Notes in Computer Science, ISSN: 1611-3349, DOI: 10.1007/11537847, Volume 3593/2005, pp. 133-143, August 29, 2005.

[100] Ralph Johnson: Design Pattern Reference, http://sourcemaking.com/design_patterns/state [accessed 16.05.09].

[101] Ralph Johnson, Design Pattern Reference: http://c2.com/cgi/wiki?StrategyPattern.

[102] Amnon H. Eden, "A theory of object-oriented design", Journal of Information Systems Frontiers, Volume 4, No. 4, pp. 379-391, 2002, ISSN: 1387-3326.

[103] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh and Eunjee Song, "A UML-Based Pattern Specification Technique," IEEE Transactions on Software Engineering, Volume 30, No. 3, pp. 193-206, March 2004, doi:10.1109/TSE.2004.1271174, ISSN: 0098-5589.

[104] Bosch Jan, "Design Patterns as Language Constructs", Journal of Object-Oriented Programming , Volume 11, No. 2, pp. 18–32, 1998, ISSN: 0896-8438.

[105] Judith Bishop, "Language features meet design patterns: raising the abstraction bar", In Proceedings of the 2nd international workshop on the role of abstraction in software engineering, pp. 1-7, Leipzig, Germany, 2008, ISBN: 978-1-60558-028-7.

[106] Lei Hu and Kamran Sartipi, "Dynamic Analysis and Design Pattern Detection in Java Programs", In Proceedings of the Twentieth International Conference on Software Engineering and Knowledge Engineering (SEKE'2008), pp. 842-846,San Francisco, CA, USA, July 1-3, 2008, ISBN: 1-891706-24-1.

[107] Tony Abou-Assalaeh and Wei Ai, "Survey of Global Regular Expression Print (GREP) Tools", March 02, 2004.

[108] Ghulam Rasool and Nadim Asif, "A Design Recovery Tool", International Journal of Software Engineering, Volume 1, No. 1, pp. 67-72 , May 2007, ISSN:1815-4875.

[109] Ghulam Rasool and Nadim Asif, "Software Artifacts Recovery Using Abstract Regular Expressions", In Proceedings of 11th IEEE Multitopic INMIC Conference, pp. 1-6, Lahore, Pakistan, December 2007, ISBN: 1-4244-1553-5.

[110] Klaus Meffert, "Supporting Design Patterns with Annotations", In Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pp. 437-445, March 27-30, 2006, ISBN: 0-7695-2546-6.

[111] Ghulam Rasool, Ilka Philippow and Patrick Maeder, "A Design Pattern Recovery Based on Annotations", In Journal of Advances in Engineering Software, Volume 41, Issue 4, pp. 519-526, April, 2010, ISSN: 0965-9978.

[112] Yann-Gael Gueheneuc, Kim Mens and Roel Wuyts, "A Comparative Framework for Design Recovery Tools ", In Proceedings of Conference on Software Maintenance and Reengineering (CSMR'06), pp.123-134, Bari Italy, March 22-24, 2006, ISBN: 0-7695-2536-9.

[113] Joppe, M. (2000), The Research Process, Retrieved February 25, 1998, from http://www.ryerson.ca/~mjoppe/rp.htm.

[114] OOPSLA Homepage: http://splashcon.org/ [accessed 16.05.09].

[115] Walter F. Tichy, "A Catalogue of General-Purpose Software Design Patterns", In Proceedings of 23rd international conference on Technology of Object-Oriented Languages and Systems, pp. 330-339, Santa Barbara, CA, USA , July 1997, ISBN: 0-8186-8383-X.

[116] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, "A System of Patterns", John Wiley & Sons, 1996, ISBN: 0471958697.

[117] James O. Coplien and Douglas C. Schmidt, "Pattern Languages of Program Design", Addison-Wesley, 1995, ISBN: 0201607344.

[118] Walter Zimmer, "Relationships between design patterns", In Proceedings of PLOP 95, pp. 346-364, 1995, ISBN: 0-201-60734-4.

[119] Vandana Bajaj,"design pattern classification": http://www.docstoc.com/docs/17376264/Design-pattern-classification.

[120] Steve Easterbrook, Janice Singer, Margaret-Anne Storey and Daniela Damian, "Selecting Empirical Methods for Software Engineering Research", Springer London 2008, ISBN: 978-1-8480-0044-5.

[121] Stephen Stelting and Olav Maassen, "Applied Java Pattern", Prentice Hall, Palo Alto, California, 2002, ISBN: 0-13-093538-7.

[122] Huston Design Patterns: http://www.vincehuston.org/dp/ [accessed 11.06.10].

[123] Design Pattern Framework 3.2: http://www.dofactory.com/Patterns/Patterns.aspx#list [accessed 16.06.10].

[124] Chanchal K. Roy, "Detection and Analysis of Near-miss Software Clones", PhD thesis, pp. 164, Queen's University Kingston, Ontario, Canada, August 2009.

[125] DPD home page: http://java.uom.gr/~nikos/pattern-detection.html [accessed 16.05.09].

[126] P-Mart Source: http://www.ptidej.net/downloads/pmart/ [accessed 16.05.09].

[127] Marek Voka, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code", IEEE transactions on software engineering, Volume 30, No. 12, pp. 904-917, December 2004, ISSN: 0098-5589.

[128] Ellen M. Vorhees, "The philosophy of information retrieval evaluation", Lecture Note in Computer Science, Volume 2406/2002, pp.143–170, January 2002, ISBN: 978-3-540-44042-0.

[130] JHotDraw Home Page: http://www.jhotdraw.org/ [accessed 22.09.09].

[131] JRefactory Home Page: http://jrefactory.sourceforge.net/ [accessed 16.10.09].

[129] Junit Home Page: http://www.junit.org/ [accessed 16.10.09].

[132] QuickUML Source: http://www.sourceforge.net/projects/quj/ [accessed 10.10.09].

[133] Apache Home Page: http://www.ant.apache.org [accessed 19.10.09].

[134] Mec Source: http://www.ibiblio.org/pub/linux/devel/debuggers/ [accessed 16.07.09].

[135] Socket Source: http://userweb.cs.utexas.edu/users/lavender/courses/socket++/ [accessed 16.07.09].

[136] The StarOffice Homepage: http://www.sun.com/software/star [accessed 09.05.10].

[137] Lajos Jeno Fulop, Rudolf Ferenc and Tibor Gyimothy, "Towards a Benchmark for Evaluating Design Pattern Miner Tools", In Proceedings of the 12th European Conference on Software Maintenance and Reengineering, pp.143-152, Athens, Greece, 1-4 April, 2008, ISBN: 978-1-4244-2157-2.

[138] Gail C. Murphy, Mik Kersten and Leah Findlater, "How Are Java Software Developers Using the Eclipse IDE?", In Journal of IEEE Software, Volume 23, No. 4, pp. 76-83, July/August 2006, ISSN: 0740-7459.

[139] Zhenchang Xing and Eleni Stroulia, "Refactoring Practice: How It Is and How It Should Be Supported—An Eclipse Case Study", In Proceedings of 22nd IEEE

International Conference on Software Maintenance, pp. 458-468, Philadelphia, Pennsylvania, USA, 24-27 September 2006, ISBN: 0-7695-2354-4.

[140] Hafedh Mili, Ali Mili, Sherif Yacoub and Edward Addy, "Reuse-Based Software Engineering Techniques, Organization, and Controls", John Wiley & Sons, 2002, ISBN: 978-0471398196.

[141] Yann-Gael Gueheneuc and Herve Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects", In proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems, pp. 296–305, Santa Barbara, California , July 2001, ISBN: 0-7695-1251-8.

[142] The Licor Homepage, http://prog.vub.ac.be/research/DMP/soul/soul2.html [accessed 15.03.10].

[143] DP-Miner tool, http://www.utdallas.edu/~yxz045100/DesignPattern/DP_Miner/ [accessed 20.01.10].

[144] IBM Rational Rose Website, http://www.ibm.com/software/rational/ [accessed 10.06.10].

[145] Dirk Beyer, Andreas Noack and Claus Lewerentz, "Simple and Efficient Relational Querying of Software Structures", In Proceedings of 10th IEEE Working Conference on Reverse Engineering, pp. 216-225, Victoria, B.C., Canada, November 2003, ISBN: 0-7695-2027-8.

[146] Mika V. Mantyla and Casper Lassenius, "What Types of Defects Are Really Discovered In Code Reviews?", IEEE Transactions in Software Engineering, Volume 35, No. 3, May/June 2009, ISSN: 0098-5589.

[147] Olivier Kaczor, Yann-Gael Gueheneuc, and Sylvie Hame, "Identification of Design Motifs with Pattern Matching Algorithms", Journal of Information and Software Technology, Volume 52, Issue 2, pp. 152-168, February 2010, ISSN: 0950-5849.

Appendix A

Abbreviations

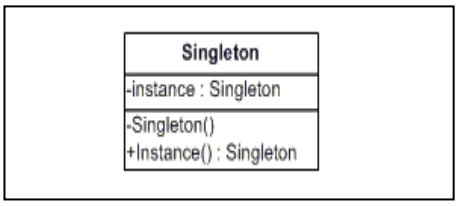
GoF	Gang of Four
AOL	Abstract Object Language
PINOT	Pattern Inference and Recovery Tool
J2EE	Java Enterprise Environment
HTML	Hypertext Manipulation Language
GUI	Graphical User Interface
UML	Unified Modeling Language
AST	Abstract Syntax Tree
SOAP	Service Oriented Architecture Patterns.
DPRT	Design Pattern Recovery Tool
DPML	Design Pattern Mining Language
RSL	Raise Specification Language
RML	Rational Manipulation Language
BPSL	Balance Pattern Specification Language
VDM	Vienna Description Method
LOTOS	Language of Temporal Ordering Specification
TLA	Temporal Logic of Actions
URN	Uniform Resource Name
OWL	Web Ontology Language
LePUS	Language for Pattern Uniform Specification
FOL	First Order Logic

PEC	Pattern Enforcing Compiler
SQL	Structured Query Language
DeIMMA	Design Motif Identification Multilayered Approach
JAPADET	Java Pattern Detector
XML	Extensible Markup Language
XMI	XML Metadata Interchange
REQL	Relational Query Language
JCL	Java Constraint Library
DFA	Deterministic Finite Automata
SCRO	Source Code Representation Ontology
RPTA	Real Time Process Algebra
SPARQL	SPARQL Protocol and RDF Query Language
RDF	Resource Description Format
SVG	Scalable Vector Graphics
API	Application Programming Interface
PTIDEJ	Pattern Trace Identification, Detection, and Enhancement in Java
DPVK	Design Pattern Toolkit
SPQR	System for Pattern Query and Recognition
GREP	Global search for regular expression and print
MVC	Model View Controller
DRT	Design Recovery Tool
XXM	eXtreme X-Machine Model
JML	Java Modeling Language
DePMoVe	Design and Pattern Modeling and Verification
PDL	Pattern Description Language

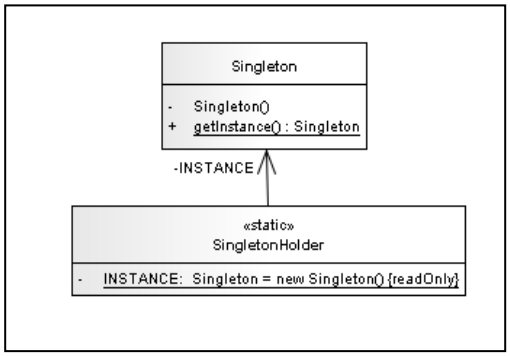
Appendix B

Features of design Patterns

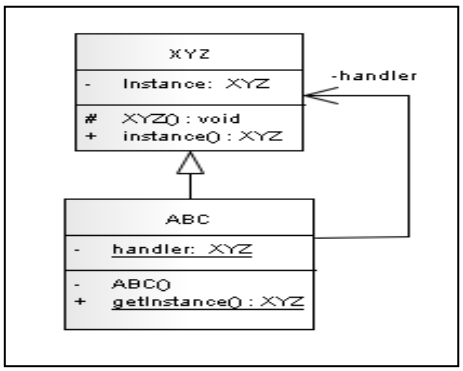
Singleton

Features	UML Structure
Has Class (Singleton) ^ IsNotAbstract (Singleton)	
Has Constructor (Singleton) ^ (is Public (Singleton) or Protected (Singleton))	
Has Static Instance (Singleton) ^ (is Public or Protected (Instance))	
Has Operation (GetInstance, Static) ^ IsPublic(GetInstance)	
Has RetrunType (GetInstance, Singleton)	
Has return Value (GetInstance, Instance) ^	
Has Annotation (A1, A2……An.) opt	

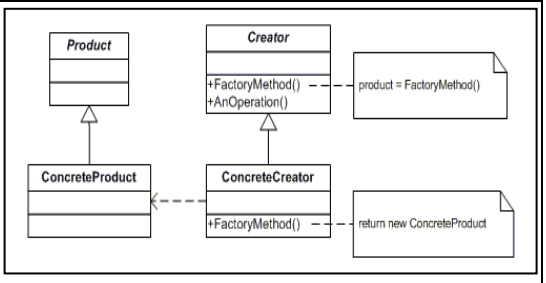
Singleton Placeholder Variant

Features	UML Structure
Has Class (Singleton) ^ IsNotAbstract (Singleton)	
Has Constructor (Singleton) ^ (is Public (Singleton) or Protected (Singleton))	
Has SingletonPlaceHolde (SingletonHolder) ^ IsPrivate (SingletonHolder) ^ IsStatic (SingletonHolder)	
Has Operation (GetInstance, Static) ^ IsPublic(GetInstance)	
Has RetrunType (GetInstance, Singleton)	
Has RetrunType (GetInstance, Singleton) ^	
Has Static Instance (SingletonHolder , Singleton) ^	
Has Annotation (A1, A2……An) opt	

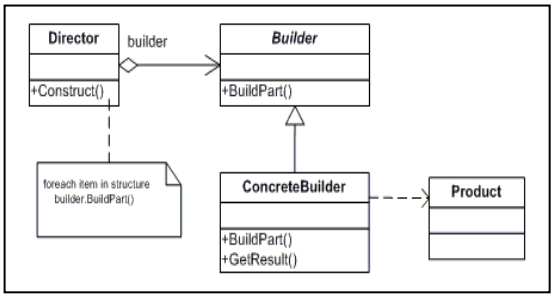
Subclass Singleton

Features	UML Structure
Has Classes (XYZ, ABC) ^ IsAbstract (XYZ)	
Has Constructor (XYZ, XYZ) ^ (is Public (Singleton) or Protected (Singleton))	
Has Operation (XYX, Instance) ^ IsPublic (Instance) ^ IsStatic (Instance)	
Has Operation (ABC, getinstance) ^ IsPublic(GetInstance) ^ IsStatic (getinstance)	
Has Inheritance(ABC, XYZ)	
Has Association (ABC, XYZ)	
Has Constructor(ABC, ABC)	
Has ReturnValue(getinstance, Instance)	
Has Annotation (A1, A2……An)opt	

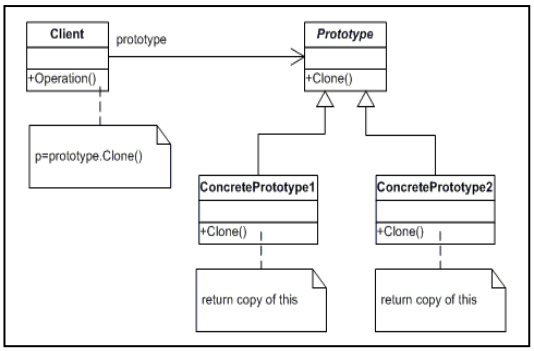
Factory Method

Features	UML Structure
Has Classes (Creator, ConcreteCreator, Product, ConcretProduct)	
Has Generalization|Realization (ConcreteCreator, Creator)	
Has Class (FactoryMehod) ^ IsAbstract (FactoryMehod)	
Has CommonOperation (Creator, ConcreteCreator, FactoryMethod Operation)	
Has Returntype(Factory Method, Product)	
Has ReturnValue (Factory Method, ConcreteProduct)	
Has Dependency (Factory Method, ConcreteProduct)	
Has Inheritance(Concrete Product, Product)	
Has NoInheritance(FactoryMehotd, Product, ConcreteProduct)	
Has Annotation (A1, A2……An) opt	

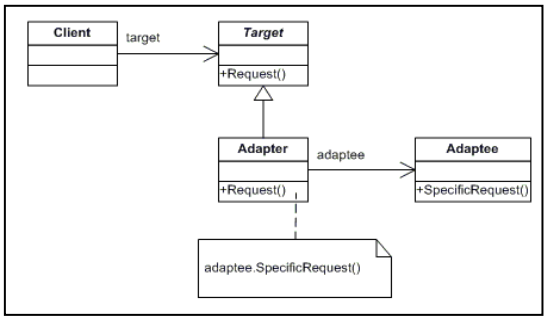
Builder

Features	UML Structure
Has Classes(Director, Builder, ConcreteBuilder, Product) ^ IsAbstract|Interface(Builder)	
Has Association (Director, Builder)	
Has Generalization(ConcreteBuilder, Builder)	
Has Delegation (Director, construct, Builder, buildpart)	
Has Instantiation(ConcreteBuilder, Product)	
Has Returnvalue(Getresult, product)	
Has Annotation(A1, A2……An)	

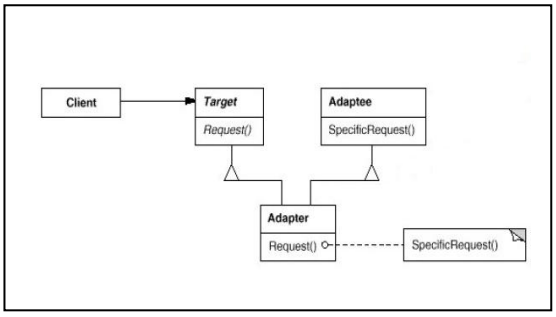
Prototype

Features	UML Structure
Has Classes (Prototype, ConcretePrototypes) ^ IsAbstract (Prototype)	
Has Generalization|Realization (ConcretePrototypes, Prototype)	
Has CommonCloneOperation (Concrete Prototype, Prototype, CloneOperation)	
Has ReturnType (CloneOperation, Object)	
Has ReturnValue (CloneOperation, T)	
Has NoInheritance|Realization (Prototype, ConcretePrototypes)	
Has Association(Client, Prototype)	
Has Delegation(Client, Operation, Prototype, CloneOperation)	
Has Annotation (A1, A2……An) opt	

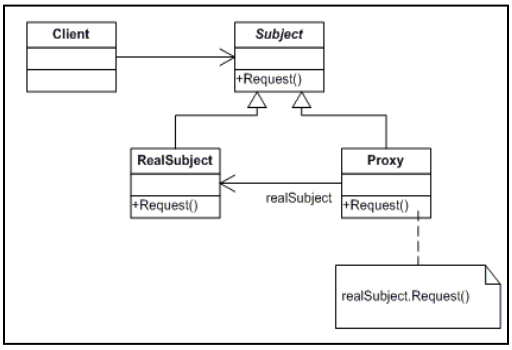
Object Adapter

Features	UML Structure
Has Classes (Target, Adapter, Adaptee) ^ IsInterface (Target)	
Has Generalization|Realization (Adapter, Target)	
Has Association (Adapter, Adaptee)	
Has Delegation (Adapter, request, Adaptee, Specific request)	
Has Operation (Adaptee, SpecificRequest) ^ IsConcrete (SpecificRequest)	
Has NoCommonInterface (Adapter, request, Adaptee, Specific request)	
HasNoDirectAccess (Client, Adaptee)	
Has NoInheritance (Adapter, Adaptee)	
Has NoInheritance (Adaptee, Target)	
Has Annotations (A1, A2……An)	

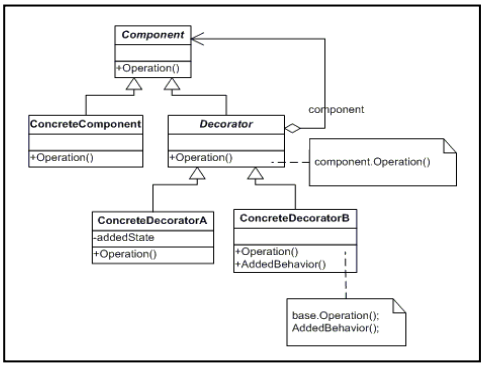
Class Adapter

Features	UML Structure
Has Classes (Target, Adapter, Adaptee) IsInterface (Target)	
Has Generalization|Realization (Adapter, Target)	
Has Generalization|Realization (Adapter, Adaptee)	
Has Delegation (Adapter, request, Adaptee, Specific request)	
Has CommonOperation (Adapter, Target, request)	
Has NoInheritance (Adaptee, Target)	
Has Annotation (A1, A2……An) opt	

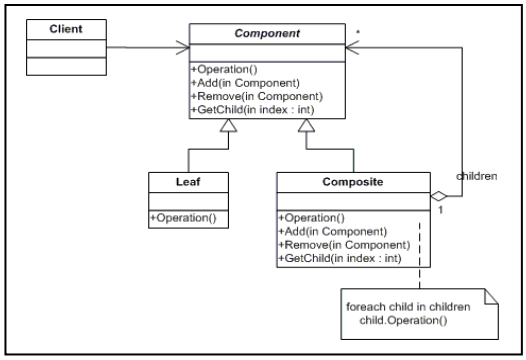
Proxy

Features	UML Structure
Has Classes (Subject, Proxy, RealSubject) ^ IsInterface (Subject)	
Has Generalization|Realization (Proxy, Subject)	
Has Generalization|Realization (RealSubject, Subject)	
Has Association(Proxy, RealSubject)	
Has Delegation (Proxy, request, RealSubject, request)	
Has CommonOperation (Proxy, Subject, RealSubject , request)	
Has Annotation (A1, A2……An) opt	

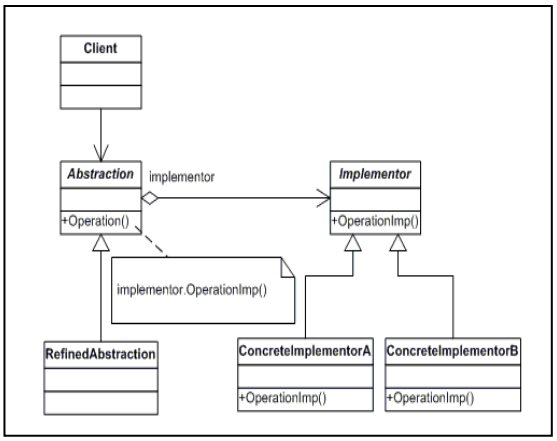
Decorator

Features	UML Structure
Has Classes (Component, Decorator, ConcreteComponent, ConcreteDecorators) ^ IsInterface|Abstractclass (Component)	
Has Inheritance (Decorator, Component)	
Has Inheritance(Concretedecorator, Decorator)	
Has Aggregation (Decorator, Component) ^ is 1-1 (Decorator, Component)	
Has Delegation (Decorator, decorate, Component, decorate)	
Has CommonOperation (Decorator, Component, ConcreteComponent, Concretedecorator, decorate)	
Has NoInheritance (Component, Decorator)	
Has NoAggregation (Component, Decorator)	
Has Annotation(A1, A2……An)opt	

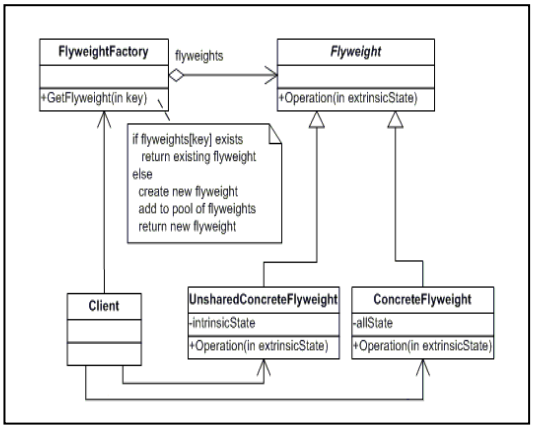
Composite

Features	UML Structure
Has Classes (Component, Composite, Leafs) ^ IsInterface|Abstractclass (Component)	
Has Inheritance (Composite, Component)	
Has Inheritance(Leaf, Component)	
Has Aggregation (Composite, Component) ^ is 1-M (Composite, Component)	
Has Delegation (Composite, Operation, Component, Operation)	
Has CommonOperations (Composite, Component, Operation, Add, Delete, GetChild)	
Has Parameters(Add, Delete, Component)	
Has NoInheritance (Component, leaf)	
Has NoInheritance (Component, Composite)	
Has Annotation(A1, A2……An)	

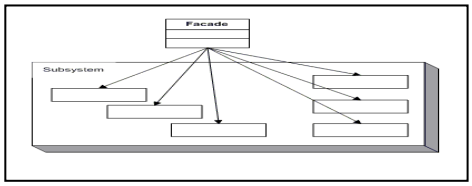
Bridge

Features	UML Structure
Has Classes (Abstraction , RefinedAbstraction, Implementor, ConcreteImplementors) ^ IsInterface|Abstractclass (Implementor)	
Has Inheritance (Abstraction, Refined abstraction)	
Has Inheritance (ConcreteImplementors, Implementor)	
Has CommonOperation (Implementor, ConcreteImplementers, OperationImp)	
Has Delegation (Abstraction, Operation, Implementation OperationImp,)	
Has Aggratation (Abstraction, Implementor) ^ Is 1-1((Abstraction, Implementor))	
Has NoInheritance (Implementor, ConcreteImplementors)	
Has NoAssociation (ConcreteImplementors, Refined abstraction)	
Has Annotation (A1, A2……An) opt	

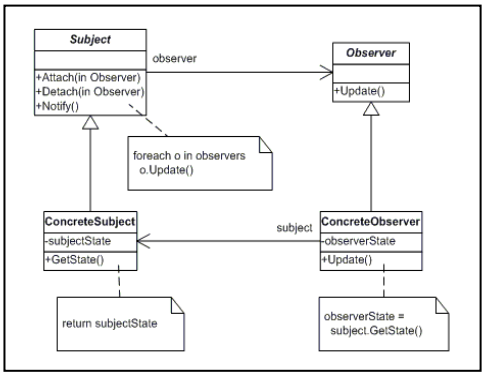
Flyweight

Features	UML Structure
Has Classes (FlyweightFactory , Flyweight, ConcreteFlyweights) ^ IsInterface|Abstractclass (Flyweight)	
Has Aggregation(FlyweightFactory, Flyweight)	
Has Operation(FlyweightFactory, Get Flyweight)	
Has Inheritance (ConcreteFlyweight, Flyweight)	
Has CommonOperation (Flyweight, ConcreteFlyweight, Operation)	
Has returnValue(Getflyweight, flyweight1|flyweight2)	
Has Association(Client, FlyweightFactory, ConcreteFlyweights	
Has Annotation (A1, A2……An) opt	

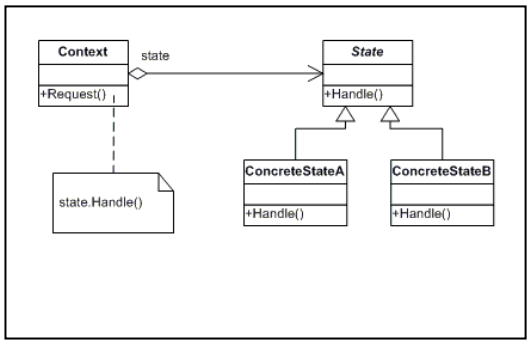
Facade

Features	UML Structure
Has Classes(Façade, SubsystemClasses)	
Has Association(Façade, SubsystemClasses)	
Has Delegation(Façade, Operation, Subsystem, Operation)	
Has Annotations(A1, A2……An)	

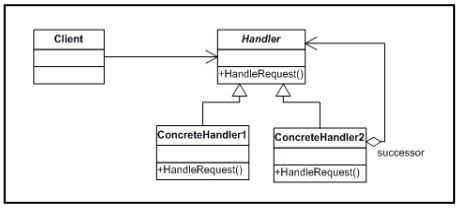
Observer

Features	UML Structure
Has Classes(Subject, ConcreteSubject, Observer, ConcreteObserver) ^ IsAbstract|Interface(Observer)	
Has Inheritance(ConcreteSubject, Subject)	
Has Inheritance(ConcreteObserver, Observer)	
Has Association(Subject, Observer)	
Has Association (ConcreteObserver, ConcreteSubject) ^ Is 1-1(ConcreteObserver, ConcreteSubject)	
Has Operations (Subject, Attach, Detach, Notify)	
Has Operation(ConcreteSubject, GetState)	
Has CommonOperation(Observer, ConcreteObserver, Update)	
Has Delegation(Subject, Attach, Observer, Add)	
Has Delegation(Subject, Detach, Observer, Remove)	
Has Delegation (Subject, Notify, Observer, Update) ^ Is 1-M (o, Update)	
Has NoAssociation(ConcreteObserver, ConcreteSubject)	
Has Annotations(A1, A2……An)	

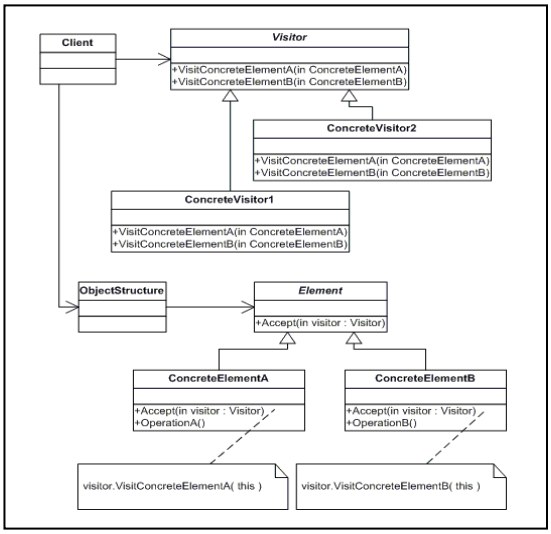
State

Features	UML Structure
Has Classes(Context, State, ConcreteStates) ^ IsAbstract|Interface(State)	
Has Inheritance (ConcreteState, State)	
Has CommonOperation (State, ConcreteStates, Handle)	
Has Delegation (Context, Request, State, Handle)	
Has NoAssociation (Context, ConcreteState)	
Has NoAssociation (ConcreteState, State)	
Has NoAssociation (ConcreteState1, ConcreteState2	
Has Call(ConcreteStateA, GetState(ConcreteStateB), Context, Request)	
Has annotations(A1, A2……An)	

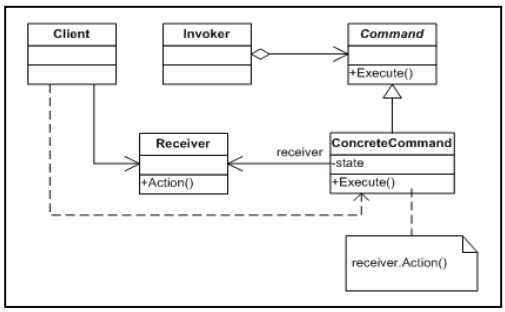
Chain of Responsibility

Features	UML Structure
Has Classes(Handler, ConcreteHandlers)	
Has Inheritance(ConcreteHandlers, Handler)	
Has Association(ConcreteHandlers, Handler)	
Has CommonOperation (ConcreteHanders, Handler HandleRequest)	
Has Delegation (ConcreteHandlers, HandleRequest, Handler, HandleRequest)	
Has Annotations(A1, A2……An)	

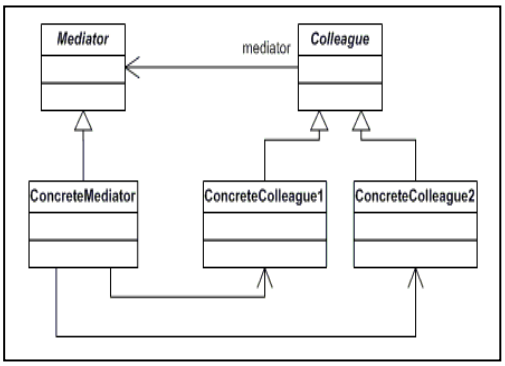
Visitor

Features	UML Structure
Has Classes(Visitor Element, ObjectStructure, ConcreteVisitors, ConcreteElements) ^ IsAbstract|Interface(Visitor)	
Has Inheritance(ConcreteVisitors, Visitor)	
Has Inheritance(ConcreteElement, Element)	
Has Association(ObjectStructure, Element)	
Has OperationWithParameter(Visitor, VisitConcreteElement(ConcreteElement))	
Has CommonOperation(Visitor, ConcreteVisitor, VisitConcreteElement)	
Has OperationWithParameter(ConcreteElement, Accept(Visitor))	
Has CommonOperation(Element, ConcreteElement, Accept)	
Has Delegation(ConcreteElement, Accept, ConcreteVisitor, VisitConcreteElement)	
Has Access(Client, Visitor, ObjectStructure)	
Has Annotations(A1, A2……An)	

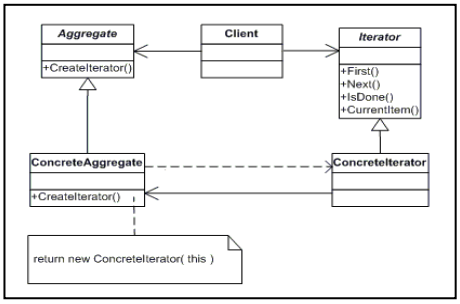
Command

Features	UML Structure
Has Classes(Invoker, Command, ConcreteCommand, Receiver)	
Has Association(Invoker, Command)	
Has Inheritance(ConcreteCommand, Command)	
Has CommonOperation(Command, ConcreteCommand, Execute)	
Has Association(ConcreteCommand, Receiver)	
Has Delegation(ConcreteCommand, Execute, Receiver, Action)	
Has Access(Client, Receiver)	
Has Annotations(A1, A2……An)	

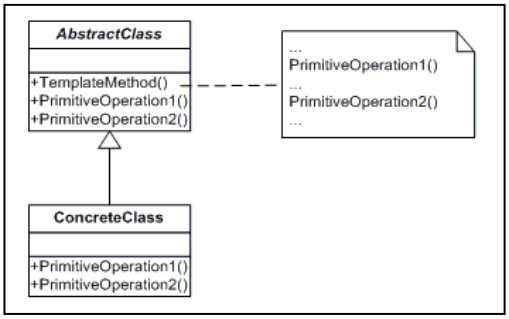
Mediator

Features	UML Structure
Has Classes(Colleague, Mediator, ConcreteColleagues, ConcreteMediator)	
Has Association(Colleague, Mediator)	
Has Inheritance(ConcreteMediator, Mediator)	
Has Inheritance(ConcreteColleagues, Colleague)	
Has Dependency(ConcreteMediator, ConcreteColleagues)	
Has Delegation(ConcreteColleague, Operation, Mediator, Operation)	
Has CommonOperation(Mediator, ConcreteMediator, Operation)	
Has Annotations(A1, A2……An)	

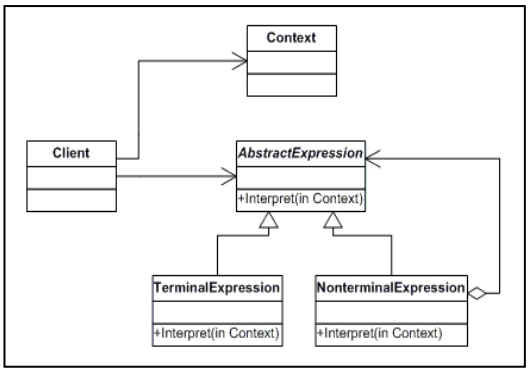
Iterator

Features	UML Structure
Has Classes(Aggregate, Iterator, ConcreteAggregate, ConcreteIterator)	
Has Inheritance(ConcreteAggregate, Aggregate)	
Has Inheritance(ConcreteIterator, Iterator)	
Has CommonOperation(Aggregate, ConcreteAggregate, CreateIterator)	
Has Returnvalue(CreateIterator, ConcreteIterator)	
Has Instantiation(ConcreteAggregate, ConcreteIterator)	
Has Annotations(A1, A2……An)	

Template Method

Features	UML Structure
Has Classes(AbstractClass, ConcreteClass	
Has Inheritance (ConcreteClass, AbstractClass)	
Has PrimitiveOperations(AbstractClass, PrimitiveOpeations) ^ IsAbstract(PrimitiveOpeations)	
Has CommonOperations (AbstractClass, ConcreteClass, PrimitiveOperations)	
Has Method Call (AbstracClass, Templatemethod, ConcreteClass, PrimitiveOperations)	
Has Annotations(A1, A2……An)	

Interpreter

Features	UML Structure
Has Classes(Context, AbstractExpression, TerminalExpression, NonterminalExpression) ^ IsAbstract|Interface(AbstractExpression)	
Has Inheritance(TerminalExpression, AbstractExpression)	
Has Inheritance(NonTerminalExpression, AbstractExpression)	
Has Aggratation (NonTerminalExpression, AbstractExpression) ^ Is 1-M((NonTerminalExpression, AbstractExpression))	
Has CommonOperation(AbstractExpression, TerminalExpression, NonterminalExpression, Interpret)	
HasOperationWithParameter (AbstractExpression, Interpret(Context))	
Has NoAssociation(TerminalExpression, AbstractExpression)	
Has Access(Client, AbstractExpression)	
Has Annotation(A1, A2……An)	

Appendix C

SQL queries and Regular Expressions

S No	Name	SQL/RegX/Parser
1	Get All Classes	select object_id from t_object where object_type='Class'
2	Get All Abstract Classes	select object_id from t_object where object_type='Class' and Abstract='0'
3	Get Interfaces	select object_id from t_object where object_type='Interface'
4	Get All Classes And Interfaces	select object_id from t_object where object_type='Class' OR object_type='Interface'
5	Has stereotype	select object_id from t_object where stereotype='%P0%'
6	Get all clonable classes	select object_id from t_object where Genlinks Like '*Implements=Cloneable;*'
7	Get name of Al lClasses	select object_id from t_object where object_type='Class' and name='%PR0%'
8	Filter Interfaces	select object_id from t_object where object_type='Interface' and object_id=%PR0%
9	Has Static instance	select t_attribute.Name from t_object, t_attribute where t_object.Object_ID = t_attribute.Object_ID and t_attribute.type=t_object.Name and t_attribute.IsStatic=1 and t_attribute.scope='private' and t_object.Object_ID=%PR0%
10	Has Static instance of Vector/List	select t_attribute.Name from t_object, t_attribute where t_object.Object_ID = t_attribute.Object_ID and (t_attribute.type='Vector' or t_attribute.type='ArrayList') and t_object.Object_ID=%PR0%
11	Has Static instance of another class	select t_attribute.Name from t_object, t_attribute where t_object.Object_ID = t_attribute.Object_ID and t_attribute.type='%PR0%' and t_attribute.IsStatic=1 and t_attribute.scope='private' and t_object.Object_ID=%PR1%
12	Has Constructor	select t_operation.Name from t_object, t_operation where t_object.Object_ID=t_operation.Object_ID and t_object.name=t_operation.name and (t_operation.scope='public' or t_operation.scope='Package' or t_operation.scope='private' or t_operation.scope='protected') and t_object.Object_id=%PR0%
13	Has operation of same class type	select DISTINCT t_operation.type from t_object, t_operation where t_object.Object_ID=t_operation.Object_ID and t_object.name=t_operation.Type and t_object.Object_id=%PR0%
14	Has operation of another class type	select t_operation.type from t_operation, t_object where t_operation.Object_ID=t_object.Object_ID and t_operation.type='%PR0%' and t_object.Object_id=%PR1%
15	Has common request operation	select F.object_id, S.object_id, T.object_id, F.name from t_operation F, t_operation S, t_operation T where F.object_id=%PR0% and S.object_id=%PR1% and T.object_id=%PR2% and F.name=S.name and F.name=T.name and F.object_id<>S.object_id and F.object_id<>T.object_id

S No	Name	SQL/RegX/Parser
16	Operation has SpecificType	select name from t_operation where type='Iterator' and object_id=%PR0%
17	Has common TM operation	"select distinct (A.name) from t_operation A, t_operation B where A.object_id =%PR0% and B.object_id =%PR1% and A.name=B.name and A.type=B.type and A.Abstract='1' and B.Abstract='0' and A.object_id<>B.object_id
18	Has common operation with same types	select A.name from t_operation A, t_operation B where A.object_id =%PR0% and B.object_id =%PR1% and A.name='clone' and B.name='clone' and A.scope=B.scope and A.type='Object' and B.type ='Object' and A.object_id<>B.object_id
19	Has abstract Operations	select name from t_operation where Abstract='1' and object_id=%PR0%
20	Has Generalisation|Realisation	select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (start_object_id=%PR0%)
21	Has Generalization	select start_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (end_object_id=%PR0%)
22	Has Generalization b/w specific classes	select start_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (end_object_id=%PR0%) and (start_object_id<>%PR1%)
23	Has Realization	select start_object_id from t_connector where (connector_type='Realisation') and (start_object_id=%PR0%" + ")
24	Has NGeneralisation	select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (start_object_id=%PR0%) and (end_object_id=%PR1%)
25	Has Association	select end_object_id from t_connector where (connector_type='Association') and (start_object_id=%PR0%" + ")
26	Has association b/w two classes	select end_object_id from t_connector where (connector_type='Association') and start_object_id=%PR0% and end_object_id=%PR1%
27	Has NAssociation	select end_object_id from t_connector where (connector_type='Association') and (start_object_id=%PR0%) and (end_object_id NOT IN (%PR1%))
28	Has association b/w specific classes	select end_object_id from t_connector where (connector_type='Association') and start_object_id=%PR0% and (end_object_id=%PR1% or end_object_id=%PR2%
29	Has Object Path	select genfile from t_object where object_id=%PR0%
30	Has hierarical Generalization	select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and start_object_id IN (select end_object_id from t_connector where (connector_type='Generalization' or connector_type='Realisation') and (start_object_id=%PR0%))
31	Has object Operations	select name from t_operation where type='Object' and name='%PR0%' and object_id =%PR1%

S No	Name	SQL/RegX/Parser
32	Operation has Parameters	select t_operationparams.Type from t_operationparams, t_operation, t_object where t_object.object_id=t_operation.object_id and t_operationparams.OperationID=t_operation.OperationID and t_operationparams.Type<>'int' and t_operationparams.Type<>'string' and t_operationparams.Type<>'Object' and t_operationparams.Type<>'boolean' and t_object.object_id=%PR0%
33	Operation has Calss as Parameters	select t_operation.name from t_object, t_operation , t_operationparams where t_object.object_id=t_operation.object_id and t_operation.OperationID= t_operationparams.OperationID and t_operationparams.type='%PR0%' and t_object.object_id=%PR1%
34	Operations has same Parameters	select A.type from t_operationparams A, t_operationparams B where A.OperationID=B.OperationID and A.type=B.type
35	Operation has Parameter value	select name from t_operationparams where type='%PR0%' and operationid=%PR1%
36	Has association label	select top_end_label from t_connector where start_object_id=%PR0% and end_object_id=%PR1%
37	Has clone operation	select name from t_operation where type='Object' and name ='clone' and object_id =%PR0%
38	Has delegation in specific class	%PR0%|%PR1%|%PR2%|%PR3%|%PR4
39	Returns all classes that are aggregated in a specific class	%PR0%|%PR1%
40	Has resolved return type	%PR0%|%P0%|%P1%|%PR1%|%PR2%
41	Has Method invocation	%PR0%|%PR1%|%PR2%|%PR3%
42	Has Method invocation through reference	%PR0%|%PR1%|%PR2%|%PR3%|%PR4%
43	Has extend|implement classes	(fiendly|public|final|abstract)?\\s*((((class)\\s*(\\w)+\\s*))((extends)\\s*(\\w)+)?\\s*((implements)\\s*(\\w)+)?(\\s*(,)\\s*(\\w+))*\\s*\\{)
44	FT9	public\\s*\\w+\\s*%PR0%\\((.*)\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*(\\w+)?\\s*\\w+\\.%PR0%\\((.*)\\)\\s*\\;
45	FT9b	public\\s*\\w+\\s*%PR0%\\((.*)\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*(\\w+)?\\s*%PR1%\\.\\w+\\((.*)\\)\\s*\\;
46	Method call	public\\s*\\w+\\s*%PR0%\\((.*)\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*%PR1%\\((.*)\\)\\s*\\;
47	Has delegation	(public|protected)?\\s*\\w+\\s*%PR0%\\(.*\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*\\w+\\.%PR1%\\(\\)\\;
48	Has return Type	(public|protected)?\\s*(\\w+)?%PR0%\\s*%PR1%\\(.*\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*(return)\\s*(new)\\s*%PR2%\\s*\\(.*\\)\\s*\\;
49	FT9a	public\\s*\\w+\\s*(\\w+)\\((.*)\\)\\s*(throws\\s*\\w+\\s*)?\\s*\\{\\s*(.)*\\s*%PR0%\\.%PR1%\\((.*)\\)\\s*\\;
50	Has comments	//+(.*))|(\/*(.*)\s(.*))
51	ImportJFile	import\s*\\w+
52	Has LoSC	LOSC-^(.*)
53	Has blank Lines	^[\\s]*$
54	Has Jave Methods	(JMethodAccessSpecifier)?\s*(JMethodModifiers)?\ \s*(JPrimTypes)?\s*(\\w+)\s*(\(\\s*(aa|zzz)?\ \)\s*((throws)\s*(\\w+))?\\s*\\{)
55	Visitor may have	Accept|class\\s*Visitor

S No	Name	SQL/RegX/Parser
56	Has Java classes	(JClasModifiers)?\\s*((Cls)((extends)\s*(\\w)+)?\ \s*((implements)\s*(\\w)+)?(\ \s*(,)\\s*(\\w+))*\\s*\\{)
57	Has Java Interfaces	((public)|(abstract))?\s*interface\s*(\\w+)\s*(extends)\s*(\\w+)\\s*\\{
58	Return Statement	return\\s*(exp)?\s*\\;
59	For statement	for\\ ((.*)\\)\\s*(block)
60	Has structure	(struct\s*\\w*\\s*\\{(Arg)*\ \s*\\})

Appendix D

Pattern Definitions Used in Prototype

Pattern	Features
Singleton A	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs4, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
Singleton B	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1e, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs11, new string[] { }, new int[] { 1, 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
Singleton C	PatternDefinitionTemp = new PatternDefinition(); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1e, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs33, new string[] { }, new int[] { 4, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));

Pattern	Features
Singleton D	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs4, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1e, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT13, new string[] { }, new int[] { 3, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
Object Adapter	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1f1, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3b, new string[] { }, new int[] { 0, 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2d, new string[] { }, new int[] { 3,2 }, true)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 6, 7, 4, 8, 5 }, false));
Class Adapter	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 7, 8, 5, 4, 6 }, false));

Pattern	Features
Proxy A	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2d, new string[] { }, new int[] { 2, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] {}, new int[] { 0}, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 7, 8, 5, 4, 6 }, false));
Proxy B	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 0,1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 1}, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 7, 8 , 5, 4, 6}, false));
Proxy C	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 0, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2b, new string[] { }, new int[] { 1, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 7, 8, 5, 4, 6 }, false));

Pattern	Features

Visitor	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1b, new string[] { }, new int[] { }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false));
	//PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT12, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1e, new string[] { }, new int[] { 3 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT13, new string[] { }, new int[] { 1, 4 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1m, new string[] { }, new int[] { 5 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 6 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 6 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 6 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT21, new string[] { }, new int[] { 9, 8, 5, 2, 1 }, false));

Factory Method	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new int[] { 0 }, true)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0,1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT19, new string[] { "true", "true" }, new int[] { 5, 4, 3 }, false));
Template Method	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5a, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4e, new string[] { }, new int[] { 0, 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); FeatureTypeInstance(FT20, new string[] { }, new int[] { 6, 5, 4, 1 }, false));

Pattern	Features

Decorator	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 0, 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 2 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 2 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 2 }, false));
	//PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT9, new string[] { }, new int[] { 4 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 7, 8, 5, 4, 6 }, false));

Composite	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTc1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT13, new string[] { }, new int[] { 2, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); //PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT9b, new string[] { }, new int[] { 4, 3 }, false));
Interpreter	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT12, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1e, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0}, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT21, new string[] { }, new int[] { 7, 6, 4, 5, 2 }, false));

Pattern	Features
Prototype	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT16a, new string[] { "'Object'" }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT16b, new string[] { }, new int[] { 1,2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
State	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1j, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 2, 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 0,3 }, true)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 3, 2 }, true)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 8,9 , 6, 5, 7 }, false)
Iterator	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT16a, new string[] { "'Iterator'" }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTc1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
Memento	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3c, new string[] { }, new int[] {0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4f, new string[] { }, new int[] { 1, 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));

Pattern	**Features**

Builder	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false));
	//PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] {0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 5, 6, 3, 2, 4 }, false));

COR	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 0, 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 1 }, false)); // path name
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false));
	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT17, new string[] { }, new int[] { 6, 7, 4, 2, 5 }, false));

Interpreter	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1j, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4d, new string[] { }, new int[] { 2, 4 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 2 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 2 }, false)); FeatureTypeInstance(FT17, new string[] { }, new int[] {8, 9, 6, 3, 7 }, false));

Pattern	Features
Observer	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1a, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTc1, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTpc, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT12, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1k, new string[] { }, new int[] { 4 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 5 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT13, new string[] { }, new int[] { 6, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT5, new string[] { }, new int[] { 5 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT21, new string[] { }, new int[] { 8, 1, 9, 10, 4 }, false));
Abstract Factory	PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1h, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1b, new string[] { }, new int[] { }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1g, new string[] { }, new int[] { 1 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs34, new string[] { }, new int[] { 2, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FTs33, new string[] { }, new int[] { 2, 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT1k, new string[] { }, new int[] { 4 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT2a, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT3a, new string[] { }, new int[] { 6, 5}, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT4z, new string[] { }, new int[] { 0, 6, 3 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 0 }, false)); PatternDefinitionTemp.AddNewFeature(new FeatureTypeInstance(FT7, new string[] { }, new int[] { 6 }, false));

Appendix E

Suggested Annotations for Documentation and Recovery of Patterns

1) @Decouple{Receiver} from {Sender}
2) @Decouple{Sender} from {Receiver}
3) @Object {Flexible Creation| Copy-Clone| Build-Compose| speed-up | control | remotely | restrict_instance_count | simplify}
4) @Compose {object} from {different_objects | related_objects }
5) @ Provide{ Same Interface| Modified Interface|Enhanced Interface| Unified Interface|Simplify Interface}
6) @Decouple {implementation} _for {dynamic variation | lists}
7) @ Wrapping{Object Variation}
8) @ Add {Functionality} at {Run time|Compile time}
9) @Construct {object} from {pieces | hierarchy}
10) @ Provide {handlers} for {requests | expressions}
11) @ Object {change} {skin | guts}
12) @Vary {algorithm} depending on {object_state | configuration}
13) @ Notify on {state_change} by {event_manager | direct_subscription}
14) @ Compose{Objects} by{Recursively |Nonrecursively}
15) @Build {Objects} by{Chain|Tree}
16) @Traverse {object_list | composite_list}
17) @ Relationship{Part-Whole} is{1-1 | 1-n| n-n}
18) @ Request {handle_by} {concrete_implementation | decoupling}
19) @State influences {algorithm | notification}
20) @State {persist | handle_discriminating}
21) @ Look for{state Change| algorithm change}
22) @Simplify {access_to} {subsystem | object_parts}
23) @Simplify {class_hierarchy} thru {access | object_creation}
24) @ Ensure {Class 1 instance| Class named M instance}
25) @ Access{elements sequentially}
26) @Unify {interface} by {handler_classes | adaptation}
27) @ Create {Object} by{Copying|Cloning}
28) @Define{1-M Dependency} between{Objects}
29) @Use {Common Interface}To{ abstract common behavior}
30) @ Encapsulate {Snapshot| Subsystem|Command}
31) @Create{Objects}
32) @ Behavior{Context dependent| State dependent+ Type Dependent }
33) @Encapsulation of{ Data| Mehod| Subclass|Another Object}
34) @Provide {(template, skeleton)} for {object_creation | algorithm| expression_parsing}

35) @Vary {independently} {abstraction_and_implementation | different_interfaces}
36) @Hide {platform_specific_classes} by {[centralizing, abstracting] | special_construction}
37) @Divide {structure} of {expression | composed_object}
38) @Vary {independently} {abstraction_and_implementation | different_interfaces}
39) @Instantiation {Lazy| Greedy Immediately}
40) @Static {class interface |method signature | public methods signature}
41) @Varies in {Interface|Type|...}
42) @Required{ Performance|Logging|Security}
43) @ Creation{Object_ Dynamically} through{Inheritance|Delegation}
44) @Exploit {common_structure} of {objects}
45) @provide {handlers} for {different_requests}
46) @Provide {Template} for{Algorithm| Production}
47) @Compose{Objects}by{recursion| other objects}
48) @Object-change {skin | guts}
49) @Vary_independently {abstraction_and_implementation | different_interfaces}
50) @ Interface{Simplify|Reuse}
51) @Interface {adapt | enhance}
52) @Share {object_instances}
53) @Allow{Undo}
54) @Treat_objects {uniformly}
55) @Unify {objects} for {object_creation | treatment}
56) @Behaviour {depends} on {state}
57) @ Dynamic Decoupled{Object Creation}
58) @ Initialization {Lazy| greedy}
59) @ Object {Encapsulation| Unification}
60) @Simplify {expression_parsing | object_creation | access_to_subsystem | list_traversal

Relationships between Annotations linked with design patterns

No	SI	FM	AF	PT	BU	AD	DE	CP	PX	FA	FL	BR	TM	CD	CR	OB	MD	IN	ST	SR	VR	IT	MM
1	x	x	x	x	x				x														
2					x		x	x															
3						x															x		
4			x							x													
5						x	x		x	x													
6		x	x		x														x				
7						x			x														
8							x																
9			x							x													
10									x							x			x				
11									x					x									x
12																			x	x			
13														x	x	x	x						
14							x	x															
15								x							x								
16		x	x	x								x			x	x	x	x		x			
17							x	x															
18															x						x		
19															x			x			x		
20					x			x											x				
21															x				x	x			
22								x														x	
23												x	x										
24	x																						
25																						x	
26						x	x		x														
27				x																			
28							x	x															
29			x																	x			
30											x			x									x

An	SI	FM	AF	PT	BU	AD	DE	CP	PX	FA	FL	BR	TM	CD	CR	OB	MD	IN	ST	SR	VR	IT	MM
31			x	x	x																		
32																x			x		x		
33							x																
34														x							x		
35						x	x		x														
36													x						x	x			
37										x	x												
38						x			x														
39	x								x														
40	x																						
41						x			x														
42									x														
43		x		x																			
44											x							x					
45									x							x			x				
46		x											x										
47							x	x															
48							x													x			
49						x						x											
50						x				x													
51						x	x	x															
52	x										x								x				
53														x									x
54	x							x			x												
55				x				x															
56																x			x				
57		x																					
58	x								x														
59								x				x											
60												x										x	

PT Prototype	**Legend**						**BR** Bridge
SI Singleton	**BU** Builder				**VR** Visitor	**DC** Decorator	**PX** Proxy
FM Factory Method	**CD** Command	**TM** Template Method	**MD** Mediator	**IN** Interpreter	**CR** Chain of Responsibility	**FL** Flyweight	**AD** Adapter
AF Abstract Factory	**SR** Strategy	**MM** Memento	**IT** Iterator	**ST** State	**OB** Observer	**FA** Facade	**CP** Composite

List of Publications

Ghulam Rasool, Ilka Philippow, Patrick Maeder, A Design Pattern Recovery Based on Annotations, In Journal of Advances in Engineering Software, ISSN(0965-9978), Volume 41, Issue 4, pp. 519-526, April 2010, Impact Factor: 1.045.

Ghulam Rasool, Patrick Maeder, Ilka Philippow, Evaluation of Design Pattern Recovery Tools, In Procedia Computer Science Journal, Vol 3, pp. 813-819, 2011.

Ghulam Rasool, Ilka Philippow, Patrick Maeder, Feature-based Design Pattern Recovery, In (SERP-2010), Software Engineering Research and Practices, pp.154-160, Las Vegas, USA, July 2010.

Ghulam Rasool, Ilka Philippow, N.A. Zafar, Recovering Design Artifacts from Legacy Applications using Automata Theory and Formal Specifications, In (SETP-2009), Software Engineering Theory and Practices, pp. 45-52, Florida USA, July 13-16, 2009.

Ghulam Rasool, Ilka Philippow, Software Artifacts Extraction for Program Comprehension, International Joint Conferences on Computer, Information, and Systems Sciences and Engineering (CISSE 2009), pp.443-447, The University of Bridgeport, USA, December 4-12, 2009.

Ghulam Rasool, Ilka Philippow, Recovering Artifacts from Legacy Systems using Pattern Matching, In Proceedings of World Academy of Science, Engineering and Technology ,Volume 36, ISSN 2070-3740, pp. 315-319, December 2008.

Ghulam Rasool, Ilka Philippow, Integrated Reverse Engineering Process Model, International Joint Conferences on Computer, Information, and Systems Sciences and Engineering (CIS2E 08), pp.307-311, The University of Bridgeport USA, December 5-13, 2008.

Ghulam Rasool, Ilka Philippow, Software Artifacts Recovery using Pattern Matching, CWRC2008, Lahore, Pakistan, pp. 58-62, June 2008.

Ghulam Rasool, Nadim Asif, Software Artifact Recovery using Abstract Regular Expressions, In proceedings of IEEE (INMIC) Conference, pp. 1-6, Lahore, Pakistan, 2007.

Ghulam Rasool, Nadim Asif, A Design Recovery Tool, International Journal of Software Engineering, ISSN (1815-4875), Vol 1, pp. 67-72 , May 2007.

Ghulam Rasool, Nadim Asif, "Software Architecture Recovery", In International Journal of Science, Engineering and Technology, ISSN (1307-6884), Vol 23, pp. 434-439, August 2007.

Curriculum Vitae

Personal Data:

Name	:	Ghulam Rasool
Date of Birth	:	08-04-1975
Nationality	:	Pakistani

Qualification:

04/2008 – 05/2011	:	**PhD** (Candidate) *(Specialization in Reverse Engineering)* TU Ilmenau, Germany.
02/2007 – 01/2008	:	**MS** in Computer Science *(Specialization in Software Engineering)* University of Lahore, Pakistan.
09/1995 – 04/1998	:	**Master** in Computer Science Bahauddin Zakariya University, Multan, Pakistan.
08/1993 – 09/1995	:	**B.Sc.** Mathematics & Physics Bahauddin Zakariya University, Multan, Pakistan.
04/1989 – 09/1993	:	**HSSC/ SSC** Certificates (Pre-Engineering) Multan Board, Pakistan.

Experience:

09/1998 – 10/1999	:	Lecturer at Army Public Intermediate Science College Gujranwala, Pakistan.
12/1999-1/2003	:	Lecturer at Petroman Training Institute Gujranwala, Pakistan.
02/2003 – 11/2007	:	Lecturer at Comsats Institute of IT Lahore, Pakistan.
12/2007- 01/2008	:	Assistant Professor at Comsats Institute of IT Lahore, Pakistan.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebnen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher order ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zu Folge hat.

Ilmenau, 04. Oktober 2010