A MULTIPLE LEVEL DETECTION APPROACH FOR DESIGN PATTERNS RECOVERY FROM OBJECT- ORIENTED PROGRAMS

MOHAMMED GHAZI KATTAR AL-OBEIDALLAH

A thesis submitted in partial fulfilment of the requirements of the University of Brighton for the degree of Doctor of Philosophy

January 2018

Table of Contents

Acknowledgments

Undertaking this Ph.D. has been a truly life-changing experience for me and it would not have been possible to do without the support that I received from many people.

I am heartily thankful to my supervisors, Prof. Miltos Petridis and Dr. Stelios Kapetanakis. Their encouragement, supervision, and support from the preliminary to the concluding level enabled me to develop an understanding of the subject. Without their guidance and constant feedback, this Ph.D. would not have been achievable.

I would like to thank my family: my parents and to my brothers and sisters for supporting me spiritually throughout writing this thesis and my life in general. In fact, acknowledge my strength, my inspiration, my guide, and my soul – my mother and my father. I offer my blessings to all of those who supported me at any level during the completion of the thesis.

Finally, this effort is dedicated to the soul of my grandmother who passed away during the completion of this thesis. May her soul rest in peace.

Declarations

I declare that the research contained in this thesis, unless otherwise formally indicated within the text, is original work of the author. The thesis has not been previously submitted to this or any other university for a degree, and does not incorporate any material already submitted for a degree.

[illegible]

Mohammed Ghazi Kattar Al-Obeidallah

Abstract

Design patterns have a key role in software development process. They describe both structure and the behavior of classes and their relationships. Maintainers can benefit from knowing the design choices made during the implementation.

This thesis presents a Multiple Level Detection Approach (MLDA) to recover design pattern instances from the Java source code. MLDA is able to recover design pattern instances based on a generated class-level representation of an investigated system. Specifically, MLDA presents what is the so-called Structural Search Model (SSM) which incrementally builds the structure of each design pattern based on the generated source code model. Moreover, MLDA uses a rule-based approach to match the method signatures of the candidate design instances to that of the subject system. As the experiment results illustrate, MLDA is able to recover 23 design patterns with a reasonable detection accuracy. Furthermore, this thesis presents a metrics-based approach to address the impact of design pattern instances on software understandability and maintainability. This approach classifies system classes into two groups: pattern classes and non-pattern classes. The experimental results show that pattern classes have better inheritance and size metrics than do non-pattern classes. Unfortunately, no safe conclusion can be drawn regarding the impact of design patterns on software understandability and maintainability, since non-pattern classes have better coupling and cohesion metrics than do pattern classes.

Abbreviations

SDLC	System Development Life Cycle
XP	eXtreme Programming
RUP	Rational Unified Process
ADD	Attribute Driven Design
GoF	Gang of Four
ISO	International Organization for Standardization
MLDA	Multiple Levels Detection Approach
SSM	Structural Search Model
AST	Abstract Syntax Tree
ASG	Abstract Syntax Graph
SPOOL	Spreading Desirable Properties into the Design of Object-Oriented, Large Scale Software
MAISA	Metrics for Analysis and Improvement of Software Architecture
CSP	Constraint Satisfaction Problem
FUJABA	From UML to Java and Back Again
AOL	Abstract Object Language
GQM	Goal Question Metric
DEPAIC++	DEsgin PAtterns Identification of C++ programs
SSA	Similarity Scoring Approach
DP-Miner	Design Patterns Discovery Matrix
CFG	Control Flow Graph
AWT	Abstract Windowing Toolkit
PTIDEJ	Pattern Traces Identification, Detection, and Enhancement
PADL	Pattern and Abstract Level Description Language
SAD	Software Architectural Defects
EPI	Efficient Pattern Identification

DRAM	Dynamic Relational Adjacency Matrix
BDD	Binary Decision Diagram
SPQR	System for Pattern Query and Recognition
PINOT	Pattern Inference and Recovery Tool
DeMIMA	Design Motif Identification, Multi-Layered Approach
DPRE	Design Patterns Recovery Environment
MARPLE	Metrics and Architecture Reconstruction plug-in for Eclipse
ARG	Attributed Relational Graph
Sempatrec	SEMantic PATtern RECovery
RDF	Resource Description Framework
SCRO	Source Code Representation Ontology
OWL	Web Ontology Language
SWRL	Semantic Web Rule Language
DL	Description Logic
CLIPS	C Language Integrated Production System
R/F generator	Rules/Facts generator
APIs	Application Programming Interfaces
CVS	Concurrent Versioning System
NOM	Number Of Methods
LCOM	Lack of Cohesion Of Methods
RFC	Total Response for Class
CBO	Coupling Between Objects
LOC	Total Lines of Code
F-IN	Fan-IN
DIT	Depth of Inheritance Tree
COH	Cohesion
FOUT	Fan-Out
NOC	Number of Children

List of Figures

List of Tables

Chapter One

Introduction

Software engineering is a branch of computer science that concerns with the design, coding and testing of software applications. Software engineering aims to improve the quality of software production by managing different software aspects such as reliability, security, dependability, portability and functionality.

The need for software engineering is increasing tremendously due to the changes in the functional and non-functional requirements. In addition, object-oriented software becomes the key element in the evolution of computer systems and the development of a high-quality software is one of the main intentions of software engineering [1].

Software engineering focuses on the behavior and specification of system structures. More specifically, software engineering is concerned with processes, methods and software tools. It involves the extraction of functional and non-functional system requirements, system architectural design, system specific design and a set of activities such as testing, deployment and maintenance.

Pressman, in his book [2], defines software engineering as "*the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines*". The Institute of Electrical and Electronics Engineers (IEEE) defines software engineering as "*the application of a systematic, disciplined and quantifiable approach to the development, operation, and maintenance of software*" [3]. Software engineering does not focus on the programming itself as the main activity. Instead, during the development process, most software engineers use ready software packages which are easy to import. The main role of a software engineer is to define "what to do" activities. In contrast, the system analyst should define "how to do" activities. The key software engineering activities are: requirements eliciting, software architecture design, coding, software integration, software deployment and software testing.

The System Development Life Cycle (SDLC) is a key concept in software engineering which involves a number of activities such as planning, analysis, design, implementation and testing. The quality of the final software system is directly affected by the earliest set of design decisions [2]. Specifically, the quality should be considered through the design, implementation and testing phases. To build and design software applications, all functional and non-functional requirements must be elicited and defined. In addition, the software architecture should be created. Software architecture involves one or more software structures which comprise elements and relationships among them. It provides a black-box representation of the software.

This chapter aims to introduce the reader to areas and aspects related to this thesis. In addition, this chapter presents our motivations for writing this thesis.

This thesis presents a design pattern detection approach to recover design instances from object-oriented programs. In addition, we tried to address the impact of these design instances on certain software quality attributes. It could be worthwhile to introduce the reader to the key concepts and aspects related to this thesis. These are: software models, software architecture, software reengineering and design patterns. Presenting these aspects helps the reader to build a general picture of the role of design patterns in the software development process.

Section One presents an overview of existing software development models. Software architecture design and documentation are presented in Section Two. Furthermore, software reengineering concepts and design patterns are presented in Sections Three and Four respectively. Section Five presents a brief overview of major software quality models. Finally, Sections Six and Seven present research questions, motivations and a list of publications made based on the thesis chapters.

1.1 An Overview of Software Development

The history of the software development process appears in Figure 1.1. Software engineering was introduced in 1968. The waterfall development model was one of the earliest developed models. Other methodologies appear continuously after the introduction of the waterfall model. In fact, the spiral development model is considered the most valuable addition to software engineering development methodologies [2], [4-6].

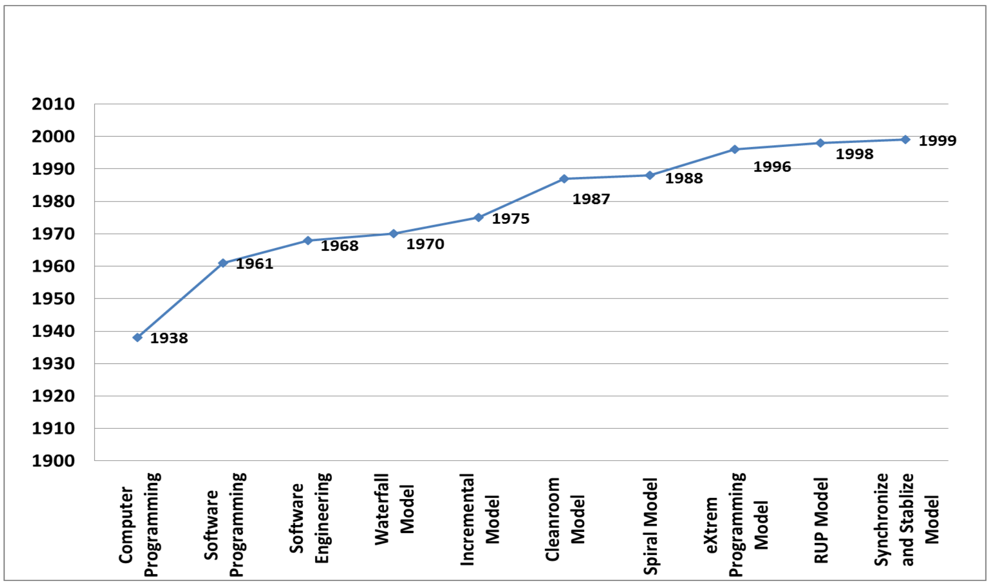


Figure 1.1: History of software development

To face changing requirement problems, extreme programming (XP) development methodology was developed by Kent Beck [2]. XP made a separation between stakeholders' decisions and business-interest decisions. Usually, XP projects start with a simple design and get the system working quickly. XP mainly relies on face-to-face communications with rare use of documentation. The XP development methodology appears in Figure 1.2 [2]. Furthermore, a widely used development methodology in many enterprises is the Rational Unified Process (RUP) [2]. It involves roles that represent the skills and responsibilities of the developers, work products, and tasks, which are activities assigned to a role. RUP works in an iterative manner and involves four main phases: inception phase, elaboration phase, construction phase and transition phase.

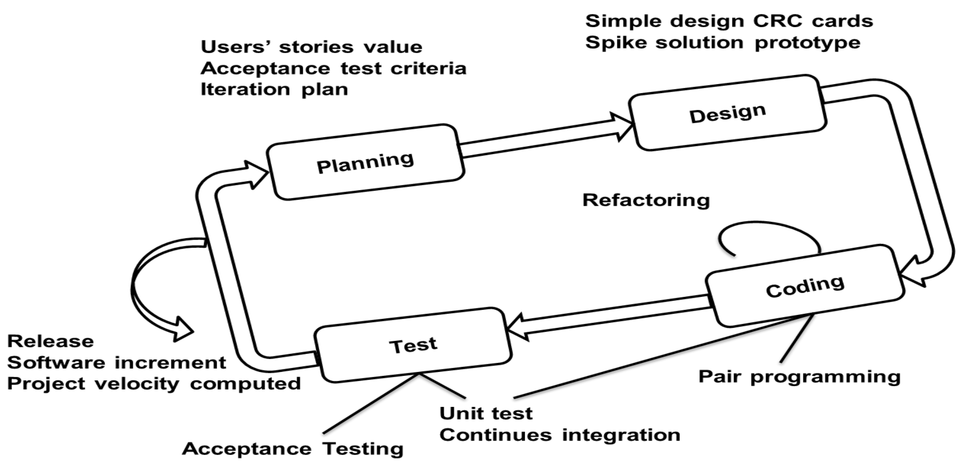


Figure 1.2: Extreme programming development methodology

1.2 Software Architecture

The software architecture is a structure, or more than one structure, which comprises elements and relationships among them [7]. The architect plays a major role in building and designing software architecture. However, different factors could affect the development of software architecture, such as stakeholders, the developing organization, architect skills and experience, and the technical environment.

Software architecture affects the earliest set of design decisions. In addition, software architecture can enable or inhibit quality attributes. The architect and the development team must decide the required quality attributes at earlier stages of the development process. For example, performance and portability often occur in mutual tension. Increasing portability by reducing the required attachments will hurt performance. Furthermore, the degree of trade-offs among quality attributes could affect the quality of the final software release.

The importance of software architecture appears clearly when investigating the software code. The architecture defines constraints on the implementation and makes the changes easier to manage. Consequently, the system will be executable early in the SDLC. Software structures which represent the main building unit of software architecture can be classified into three categories based on the nature of the elements that they show [7]:

- Module structure: units of code are the main building elements of this structure. The decomposition structure, class structure and uses structure are a few examples of module structure.
- Component and connector structure: elements were represented as components, whereas the relationships between them were represented as connectors. Client-server and concurrency structures are two popular examples of a component and connector structure.
- Allocation structure which shows the relationship between software elements and other external elements. Work-assignment structure and deployment structure are two popular examples of an allocation structure.

1.2.1 Software Architecture Design

Developing software architecture's design requires the elicitation of quality and functional and business requirements. The design of a software architecture should provide a black-box representation of a subject system since no details are required at this design level. The Attribute Driven Design (ADD) is one of the most popular methods for software architecture design. ADD has been developed by extending other design development methods [7].

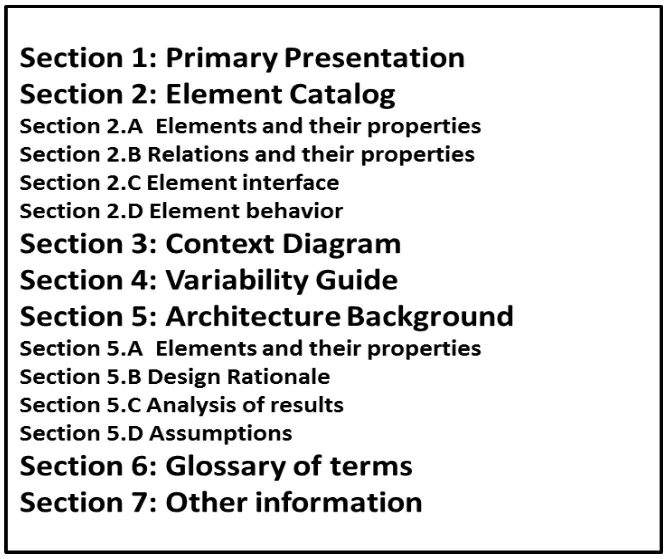
ADD uses quality attribute scenarios as input and produces levels of a module decomposition view as output. ADD starts by selecting the module to decompose (i.e. the whole system). Then the system will be decomposed into a number of sub-systems which will also be decomposed in their turn into sub-modules. Based on the quality attribute scenarios and the functional requirements, architectural drivers will be selected. The next step in designing the architecture is the selection of suitable architectural designs (or, in certain cases, creating them). Finally, for each module, the instantiation and the assigning of functionalities were performed. The implementation of quality attributes during software architecture design helps the development team to differentiate a good system from a bad one.

1.2.2 Software Architecture Documentation

Documenting a software architecture helps the development team to decide what it is important to capture. The documentation process relies on the collected requirements and should be abstract enough to be understood by the stakeholders.

To create a software architecture's document, the architect and the development team should select the relevant views (the view is a representation of the software architecture), document a view and then document the information that fits more than one view [7][8], consequently producing a list of candidate views. In fact, some of the views can fit every system. Hence, the number of relevant views is quite large. To reduce the list size, views that serve very few stakeholders can be merged with a view that gives information from two or more views at once. Finally, the list of the views is prioritized based on the project details. Unfortunately, there is no standard template to document views. However, the seven-part template presented by Bass *et al.* [7] shows good results in practice. The seven parts of a documented view appear in Figure 1.3.

The first section of the template is the primary representation. This section shows the elements of the architecture and their relationships. Graphical representations and tables were used to represent these elements and their relationships. The details of the elements and their relationships appear in the element catalog section. Moreover, the element catalog section shows the behavior and interface of each element. The context diagram section shows the relationship between the system and its environment. The variability guide section helps the architect and the development team to trace any variation points. The type of variation point depends on the type of view. In the case of module views, the variation points are versions of the modules. On the other hand, the variation points are constraints on the scheduling in a component and connector view, whereas they are conditions on the view in an allocation view.



Section 1: Primary Presentation
Section 2: Element Catalog
Section 2.A Elements and their properties
Section 2.B Relations and their properties
Section 2.C Element interface
Section 2.D Element behavior
Section 3: Context Diagram
Section 4: Variability Guide
Section 5: Architecture Background
Section 5.A Elements and their properties
Section 5.B Design Rationale
Section 5.C Analysis of results
Section 5.D Assumptions
Section 6: Glossary of terms
Section 7: Other information

Figure 1.3: The seven parts of a documented view

The architecture background section explains why the design has been made in this way. It shows how the assumptions and the analysis results were reflected in the design. The glossary-of-terms section shows the terms used in the views. The last section is the other information section which shows different kinds of information, such as management and decision support information.

The documentation of a view is completed by collecting the information that applies to more than one view. This can be done by developing a cross-view documentation template which shows how a document is organized.

1.3 Software Reengineering

Reengineering is a rebuilding activity that improves the maintainability and structure of software systems. Reengineering may involve modifying and updating a system's internal data without affecting its functionality and/or its architecture.

Moreover, reengineering provides a cost-effective development solution. Usually, the costs of reengineering current software are less than the costs of building a new one. Chikofsky and Cross [9] introduced the concept of forward engineering to describe conventional software development. Forward engineering uses a system specification as a starting point.

In contrast to forward engineering, reengineering [2] starts with an existing system. The quality of software to be developed, the involvement of expert staff and data size can affect the reengineering process. The key activities of a reengineering process are presented in Figure 1.4. The process model of software reengineering is cyclical; this means that each activity may be revisited more than once. The activities are performed in a linear manner.

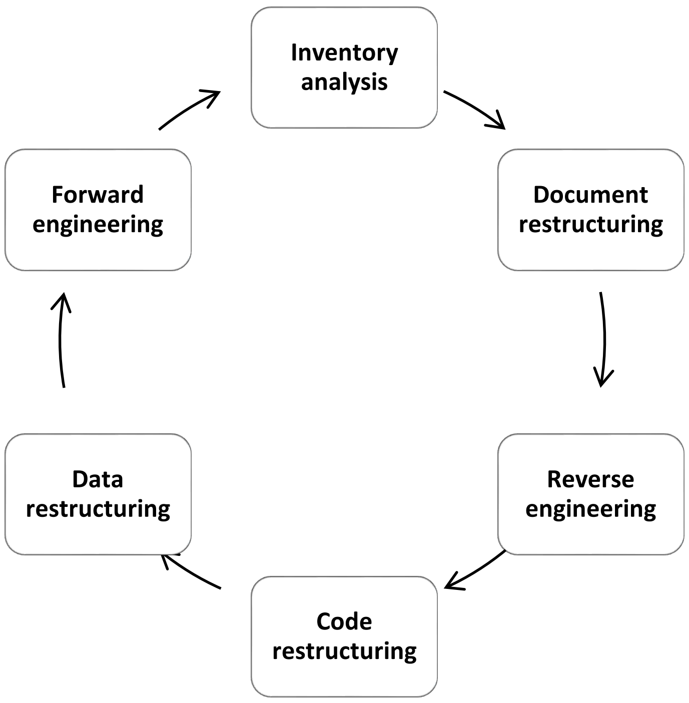


Figure 1.4: The process model of software reengineering

The activities of the software reengineering process model can be described as follows:

- Inventory analysis: the inventory is a spreadsheet model that provides certain details such as the critical business and size of every active application. The candidates for reengineering can be achieved at this stage by sorting the detailed information, taking into account the maintainability aspects.

- Document restructuring: restructuring of a document is considered one of the key reengineering activities. Weak documentation affects the quality of the final specification. In addition, the recreation of the documentation puts too much of a burden on the development team.
- Reverse engineering: reverse engineering is the process of design recovery which extracts design information from the source code. Reverse engineering normally uses object models, data models, UML classes, state diagrams and deployment diagrams. The level of abstraction issues, completeness issues and directionality issues must be considered during any reverse engineering activity. The completeness of the reverse engineering process is determined by the level of details at any abstraction level [2]. The abstraction level depends on the complexity of the recovered design. The completeness and the level of abstraction often occur in mutual tension. All recovered information from the source code could be used later during the maintenance activities.
- Code restructuring: code restructuring aims to produce a high-quality design that provides a functionality similar to that of the original source code.
- Data restructuring: data can be restructured to extract certain data items and objects. Different kind of statements (I/O statements, data-definition statements etc.) should be evaluated. At this stage, data re-design is performed by the so-called "data name rationalization" to ensure that all naming conventions fit local standards.
- Forward engineering: Forward engineering uses software engineering methods and concepts to recreate an existing application. Usually, the newly developed application has more capabilities than does the

original application. Forward engineering can be applied to client/server architectures and to object-oriented architectures.

Object-oriented development becomes the main development paradigm in many software enterprises. The conventional software is reverse engineered into an object-oriented implementation by creating appropriate data, functional models and behavioral models. Sub-systems and class hierarchical and object models should be created as well. Some classes must be engineered from scratch and redesigned to fit the new object-oriented architecture.

1.4 Design Patterns

The concept of patterns was introduced into the field of architecture by Christopher Alexander who documented reusable architectural proposals for producing high-quality designs [10].

In software engineering, a pattern is a recurring solution to a standard problem in a context. In 1995, the idea of patterns was adopted by the so-called Gang of Four (Gamma, Helm, Johnson and Vlissides) [11] -henceforth GoF. The GoF cataloged 23 design patterns. Each design pattern describes a problem that occurs over and over again in an attempt to describe the core solution to that problem. This solution can be re-used a million times over without doing it the same way twice. In fact, design patterns vary in their levels of abstraction. Each design pattern solves a specific design problem by connecting together a number of classes (participant classes) using different relationships. According to the GoF's catalog, each design pattern involves both structural and behavioral aspects. Structural aspects describe the static arrangement of classes and their relationships. On the other hand, behavioral aspects describe dynamic interactions between pattern participant classes.

Design patterns at the source code level reflect the earliest set of design decisions taken by the development team. In addition, the majority of current software systems involve instances of design patterns in their source codes. Design patterns can improve software documentation, speed up the development process and enable large-scale reuse of software architectures. The template of design patterns presented by the GoF involves several sections, such as participants, structure and collaboration sections. These sections describe the design fingerprint (motif) [11]. Each occurrence of a design pattern in the source code is called a design pattern instance. A design instance should reflect the required pattern structure and behavior presented by the GoF. However, design pattern instances could be partially implemented in the source code (i.e. one or more of the instance's participant classes are not implemented). A participant class is a class that plays a certain role in design pattern instance.

Many approaches and tools were introduced in the literature to recover GoF design instances from object-oriented programs. Most of these approaches recover a few patterns and they lack accuracy. This thesis tries to recover all GoF design patterns with a reasonable detection accuracy by presenting a structural rule-based approach. The presented approach covers the structural and dynamic aspects of GoF design patterns.

Although GoF design patterns become standard designs for software development, they are not only the available design templates. For example, other design templates use architectural patterns such as model view controller, idiom patterns and gaming patterns. This thesis focuses on GoF design patterns. This does not suggest that GoF patterns are better than other design patterns, but all primary studies deal with GoF design patterns.

In addition to design patterns recovery, the quality of software systems is another key aspect related to design patterns. The quality of software systems could be affected positively, negatively, or not affected at all, when certain design patterns are implemented. In this thesis, a metrics-based approach has been developed to address the impact of design pattern instances on certain quality attributes. This approach uses the detected design pattern instances that have been recovered by the structural rule-based approach.

1.5 Software Quality Attributes

Measurement plays an important role in evaluating the quality of software applications. Quality is "the conformance to the functional and non-functional requirements, development standards and expected software characteristics" [12].

A number of quality models have been developed to assess and characterize the quality of software systems. Each model has its own elements, characteristics and attributes and most of these quality models quantitatively measure the internal aspects of software applications. Followings are the most widely used quality models.

1.5.1 McCall’s Quality Model

Factors that affect the quality can be measured directly or indirectly (e.g. usability). McCall *et al.,* henceforth McCall’s quality model, introduced a number of useful factors that affect the quality of software applications. These factors focus on the product’s operational characteristics, the ability to accept changes and the flexibility to work in new environments [12].

McCall’s quality model is considered a predecessor of today’s quality models. It tries to bridge the gap between users and developers by introducing quality factors that reflect their views. More specifically, McCall’s quality model involves 11 quality factors to describe the user’s view (external view), 23 quality criteria to describe the developer’s view (internal view) and metrics, which provide a method for measurement. The metrics were achieved by answering a set of yes-and-no questions. Consequently, the quality is subjectively measured based on the person(s) who answer the questions. McCall's quality model is presented in Figure 1.5.

1.5.2 Boehm’s Quality Model

Boehm’s quality model qualitatively defines software quality using a set of metrics and attributes [13]. Boehm’s model involves three characteristic levels: high, intermediate and primitive.

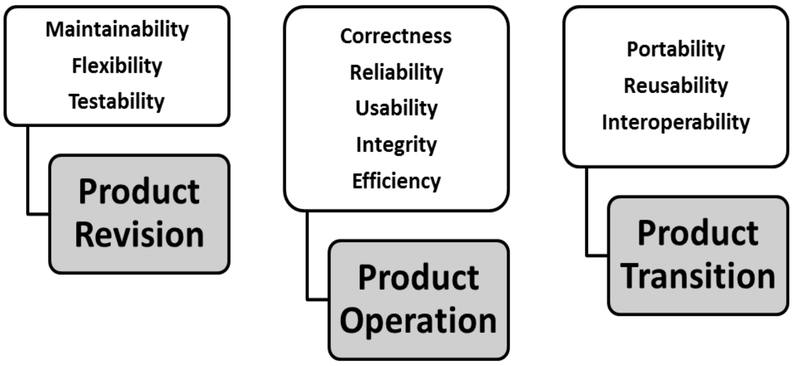


Figure 1.5: McCall's quality model

The high-level characteristics handle the as-is utility, portability and maintainability aspects. On the other hand, the intermediate-level characteristics involve seven quality factors that represent the expected qualities from the application (i.e. portability, reliability, efficiency, usability, testability, understandability and flexibility). Boehm's quality model appears in Figure 1.6.

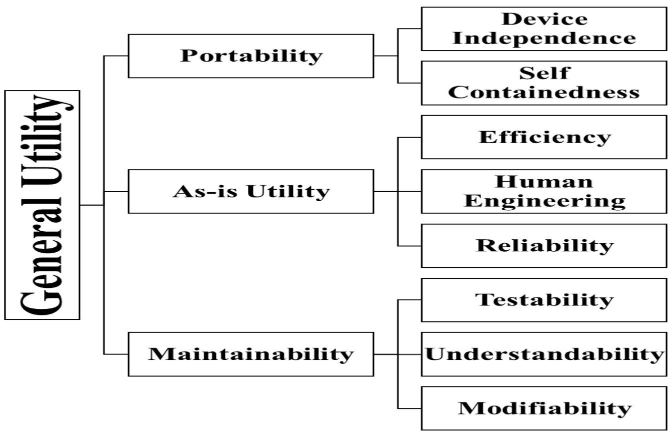


Figure 1.6: Boehm's quality model

1.5.3 ISO Quality Model

ISO stands for International Organization for Standardization. The ISO 9001 family is the most widely spread and widely used family. The ISO 9001 management system handles the internal and external aspects of the organization by managing the quality of the delivered products. In addition, ISO 9001 provides design, implementation, support and documenting activities for quality and resource management processes, market and regularity research processes, product protection processes, customer needs, communication processes and training processes [17]. Based on McCall's quality model and Boehm's quality model, the ISO has released the ISO/IEC 9126 version to identify the internal and external quality characteristics. The identified characteristics are:

- Functionality: the degree to which the software works as intended.
- Usability: how easy it is for a user to accomplish a desired task.
- Reliability: the degree to which the software delivers its services when needed.
- Efficiency: the degree to which the software uses system resources in an optimal way.
- Portability: the software application is portable when extracted from its development environment and transported easily to another environment.
- Maintainability: the degree to which the software is easy to repair.

Figure 1.7 shows the ISO/IEC 9126 quality attributes and their sub-attributes:

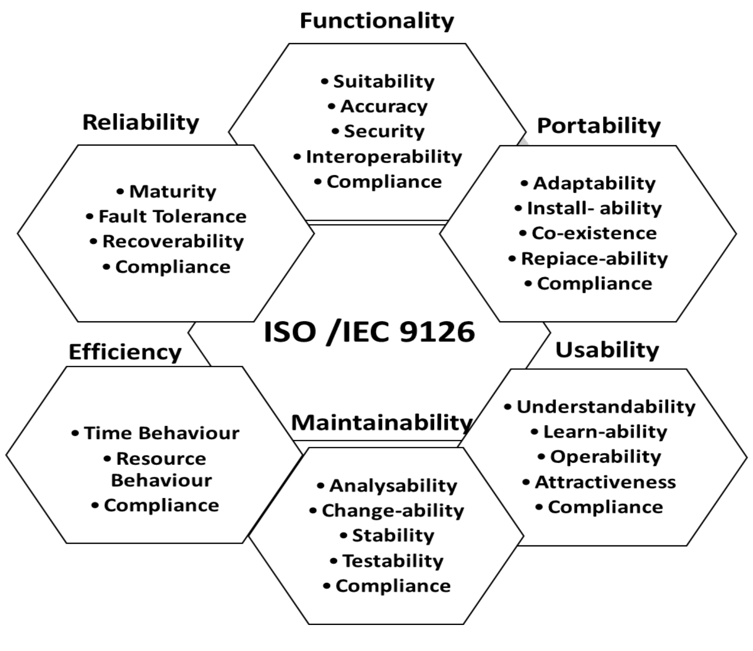


Figure 1.7: ISO/IEC 9126 quality attributes

In addition, ISO has released the 15504 version to address all processes involved in supporting, supplying, maintaining, operating, developing and delivering software activities. This thesis relies on the ISO/IEC 9126 as a reference model when describing and/or using any of the quality attributes. ISO/IEC 9126 is a widely accepted and used quality model in the software engineering community.

1.5.4 Other Quality Models

Other quality models were presented to characterize the quality of software systems. The FURPS model has been developed by Robert Grady [15] and extended by Rational Software Company [16] to FURPS+ to include design, implementations and interface requirements. FURPS stands for Functionality, Usability, Reliability, Performance and Supportability. Furthermore, the quality model developed by Dromey [17] attempts to handle the problem of evaluation differences between software products. Dromey's model has been applied to different systems and involves three basic elements: software product, product properties and quality attributes. In addition, Dromey's quality model attempts to link the properties of a software product to software quality attributes. This can be achieved by selecting high-level quality attributes, listing system components/modules, investigating the effects of each property on the quality attributes and evaluating the model.

1.6 Motivation

This section presents the key motivations for building a new detection approach to recover design pattern instances from object-oriented programs. In addition, the general motivations behind the development of a metrics-based approach to address the impact of design pattern instances on software quality attributes are presented.

Recovering design pattern information from a source code helps to document the systems, enable large-scale reuse of software architectures and speed up the software development process. However, current design pattern detection approaches lack accuracy and recover a few design patterns. Only three approaches recover ALL GoF design patterns. Moreover, most detection approaches focus on the structural features of design patterns. Specifically, these approaches focus on four relationships that may occur between classes and interfaces inside any object-oriented program (inheritance, aggregation, association and dependency). Hence, most current detection approaches fail to recover all the possible structures similar to that of ALL GoF design patterns since these approaches search for a maximum of four relationships and their matching techniques are not able to accurately match the recovered structures to that of GoF (e.g. matrices matching using similarity scoring and exact matching, sub-graph discovery, machine learning and software metrics, data flow analysis, constraints over variables, class diagram analysis and semantic rules). This introduces the necessity of a new recovery approach that can recover all the possible structures similar to that of ALL GoF design patterns.

In addition, current recovery approaches ignore the role of method signatures of GoF design patterns during the recovery process. These approaches face difficulties in creating a suitable representation of the method signatures and match them to that of GoF. Hence, these approaches only recover a few patterns and their accuracy is not reasonable. Consequently, a new recovery approach should be developed to accurately recover ALL GoF design patterns.

The new recovery methodology should cover both the structural and behavioral aspects of GoF design patterns. The structural aspects should reflect the required structure (arrangement of classes) presented by GoF, whereas the behavioral aspects should reflect the required interactions between pattern participant classes.

This thesis presents a rule-based approach to handle the problem of representing the required method signatures of GoF design patterns. More specifically, a rules template has been created to reflect the required method signatures of the candidate design pattern instances. The main motivation of using a rule-based approach is its ability to represent the method signatures of the candidate design instances as an independent piece of knowledge, which can be transformed into a set of rules. In addition, the method signatures of GoF design patterns have a uniform structure which facilitates their representation as a set of rules.

In fact, the motivation of the work in this thesis is to explore the effects of representing the method signatures of GoF design patterns using a rule-based system on the process of design patterns recovery which relies on relationships matching. As a result, a novel methodology for design patterns recovery is proposed. This methodology combines both structural and behavioral features of GoF design patterns.

Furthermore, this thesis tries to address the impact of GoF design patterns on software maintainability and understandability since they are the most commonly investigated quality attributes in the available literature. The impact of GoF design patterns on software quality attributes provides a support for decision-making during software design and refactoring. Most previous studies in the literature used experiment, case studies, conceptual analysis and survey to assess the impact of design patterns on software quality attributes. We believe that these methods lead to controversial results since most of them are based on human intervention and lack accuracy. Consequently, there is a necessity to introduce a new approach to address the impact of GoF design patterns on the quality of software systems. This thesis presents a metrics-based approach to assess the impact of GoF design patterns. The main motivation behind the use of software metrics to quantify the subject system is their ability to provide a static and stable representation of the subject system. Moreover, since our proposed recovering methodology relies on a class-level representation of the subject system and the recovered design instances have been validated based on the all publicly available results in the literature, a list of classes playing roles in design patterns was generated for all subject systems. This will facilitate the use of class-level metrics to quantify each class in the subject system. Hence, software metrics can be calculated for classes playing roles in design patterns and can be compared with classes that don’t play roles in design patterns.

1.7 Aims and Objectives

This thesis mainly aims:

- To recover ALL GoF design patterns with reasonable detection accuracy based on hybrid structural and behavioral characteristics.

Hence, a structural search model which relies on the principle of relationships matching has been developed to reflect the required structural aspects. In addition, the method signatures of GoF design patterns have been represented using a rule-based approach to reflect the required behavioral aspects.

The structural search model will recover the instances of GoF design patterns based on the relationships matching between classes and interfaces. Furthermore, the rule-based approach will be applied to the recovery process. This leads to the second aim of this thesis:

- To explore the effects of applying a rule-based approach to the process of design patterns recovery which relies on the relationships matching.

Finally, this thesis aims:

- To address the impact of GoF design patterns on software maintainability and understandability.

1.8 Research Questions

The main research questions formulated in this thesis can be stated as follows:

- Is the Structural Search Model (SSM), which relies on the relationships matching, able to recover instances of design patterns with a reasonable detection accuracy?

The structural characteristics of GoF design patterns will be recovered using the developed Structural Search Model. This model relies on the relationships matching between classes and interfaces and builds the structure of ALL GoF design patterns incrementally. In addition, the SSM will try all the possible combinations between classes and interfaces to recover all the possible structures similar to that of GoF.

- Is the use of a rule-based system to match the method signatures of the candidate design instances to that of a subject system able to reduce the number of false positive candidate instances (i.e. enhancing the detection accuracy of design patterns which relies on relationships matching)?

The behavioral characteristics of GoF design patterns will be represented using a rule-based approach. A rule template has been created to represent the required method signatures of GoF design patterns. The rule-based approach will be applied to filter the candidate design instances recovered by the SSM.

- Do classes playing roles in design patterns have better software metrics than other classes in the system (i.e. do design pattern instances enhance software maintainability and understandability)?

The third research question tries to address the impact of design pattern instances on certain quality attributes. More specifically, we tried to address the impact of design instances on software maintainability and understandability since they are the most commonly investigated quality attributes. All subject system classes will be classified into two groups: classes that are playing roles in design patterns and classes that are not. In addition, software metrics, with their correlations to quality attributes, will be used in attempts to reach a safe conclusion.

1.9 Contribution to Knowledge

The main contribution of this thesis is the addition to the body of software engineering of real evidence on the effects of applying a rule-based approach to the process of design pattern detection, which relies on relationships matching.

A Structural Search Model (SSM) has been developed to recover the instances of GoF design patterns based on the connecting relationships between classes and interfaces. This model builds the structure of each design pattern incrementally, tries all the possible combinations between classes and interfaces inside a subject system and relies on the five key relationships that may occur between these classes and interfaces (Inheritance, Realization, Aggregation, Association and Dependency). On the other hand, A rule-based approach to filter the candidate design instances recovered using the SSM has been developed. This rule-based approach matches the method signatures of GoF to that of the subject system. Moreover, this thesis proposes a rules template to reflect the required method signatures of GoF design patterns.

Consequently, the hybrid structural rule-based approach tries to achieve reasonable detection accuracy in terms of recall and precision. This approach combines both structural and behavioral characteristics of GoF design patterns.

Furthermore, the second contribution of this thesis is to address the impact of GoF design patterns on software maintainability and understandability. A metrics-based approach has been developed in an attempt to reach a safe conclusion.

1.10 Thesis Organization

This thesis is organized as follows: Chapter Two highlights and compares most of the detection approaches presented in the literature In addition, Chapter two presents the reported impact of GoF design patterns on software maintainability and understandability. The general methodology and the structural search model to recover design pattern instances are discussed in Chapter Three. Chapter Four presents the experimental results of applying the structural search model to eight subject systems. The rule-based system and its effect on detection accuracy are presented in Chapters Five and Six respectively. Chapter Seven addresses the impact of design patterns on software understandability and maintainability. Finally, thesis conclusions, future research directions and limitations of the presented work are presented in Chapter Eight.

1.11 Summary

This chapter presented an abstracted overview of areas and aspects related to this thesis. These aspects are software development models, software architecture, design patterns and quality attributes. Moreover, the key motivations for writing this thesis and these research questions, aims and contribution to knowledge were presented.

Chapter Two
Literature Review

Since the introduction of design patterns in 1995, many tools and approaches have been presented in the literature to recover their instances from object-oriented programs. This chapter provides the current state of the art in design pattern detection. In addition, the reported impact of design patterns has been presented. The selected approaches cover the whole spectrum of the research in design pattern detection. We noticed diverse accuracy values recovered by different detection approaches and the lessons learned are listed at the end of this chapter. These can be used for future research directions and guidelines in the area of design pattern detection.

2.1 Introduction

Many tools and approaches have been developed in the last two decades to recover design pattern instances from object-oriented programs. The main objective of these approaches is to recover accurately the instances of design patterns. However, detection approaches differ in their input, extraction methodology, case studies, recovered patterns, system representation, accuracy and validation method.

The field of design pattern detection still faces a number of key challenges, such as the fact that the current detection approaches are working independently from each other and there are no standard benchmarks or references to validate the recovered instances and the possible variants of the design pattern. In addition, the evaluation of design pattern detection approaches is somehow difficult since most current detection tools are not publicly available. Some detection approaches applied their experiments on small-size programs using a few patterns and these achieved high precision and recall rates. However, most detection approaches rely on code level for patterns detection, using the source code as input and representing it in one of the parsing formats, Abstract Syntax

Tree (AST) or Abstract Syntax Graph (ASG). Moreover, the current design pattern detection tools are built to implement a certain methodology and each tool works independently without the ability to integrate it with other existing tools.

The accuracy of design pattern detection tools is affected by a number of factors, such as pattern variants, instance definition, type of matching, system size and parsing and modeling techniques. In fact, most detection approaches recover only one category of design patterns or only certain patterns.

On the other hand, the implementation of design patterns can vary across studies and these variants could be responsible for any differences observed in the reported results of the effects of design patterns on quality attributes.

An empirical review and evaluation of current existing detection approaches is important to guide the researcher through the weaknesses of these approaches. This chapter presents the current state of design pattern detection approaches. Specifically, we have presented a comparative study on design pattern detection approaches in terms of detection methodology, analysis style, system representation, subject systems, recovered design patterns and evaluation criteria. Furthermore, this chapter presents the reported impact of design patterns on software quality attributes.

2.2 Overview of Detection Approaches

Design pattern detection approaches could be classified based on different criteria and aspects. This chapter has categorized detection approaches based on their detection methodology and on their analysis style.

Most detection approaches use similar key steps which aim to match the source code representation to that of GoF. The detection methodologies could be categorized, based on their key recovery steps, into four main groups: database-query approaches, metrics-based approaches, UML structure, graph and matrix-based approaches and miscellaneous approaches.

2.2.1 Database-Query Approaches

Database-query approaches transform the source code into an intermediate representation, such as AST, ASG, UML structures, XMI etc. SQL queries are used to recover pattern information from the generated representation. The database in use affects the performance of the queries. Unfortunately, database-query approaches are not able to recover instances of behavioral design patterns.

2.2.1.1 Rasool et al. Approach

The approach presented by Rasool *et al.* [18] used annotations, regular expressions and database queries to recover instances of design patterns. The varying features of patterns are defined and rules are applied to match these features to source code elements. Time and search space are reduced by using appropriate semantics from large legacy systems. Rasool *et al.*'s approach only recovers certain patterns and its accuracy and efficiency are not reported.

2.2.1.2 Stencel and Wegrzynowicz Approach

Stencel and Wegrzynowicz [19] present a pattern recognition method to detect non-standard implementations of design patterns as well as standard implementations. The Detection of Diverse Design Pattern Variants tool (D3) has been developed to implement the detection methodology. In addition, a simple program meta-model has been generated to store the program's core elements, such as attributes, operations and instances. D3 detected the creational instances of design patterns from the Java source code using static analysis and SQL queries. The execution time is reported only where D3 spent 36 seconds to recover the creational instances from JHotDraw.

2.2.1.3 Marek Vokac Approach

Marek Vokac constructed a tool to recover certain design patterns from the C++ source code [20]. The tool relies on descriptions of structural signatures associated with the chosen design patterns. The UNDERSTAND FOR C++ parser [21] has been used to generate a file that stores entities' and references' data which will be transferred into an SQL database. The SQL table involves links to certain files and metrics. The recognition of design patterns is done by a series of SQL statements designed to look for a given structure. The experiments were conducted only on a Customer Relationship Management system.

2.2.1.4 SPOOL

SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) is a joint research project between the University of Montreal and Bell Canada. The SPOOL environment comprises functionality for design composition, change effect analysis and detection of design patterns [22]. The SPOOL reverse engineering environment involves a three-tier architecture. The top tiers involve source code capturing, analysis and visualization. On the other hand, the design repository resides on the bottom tier. The SPOOL environment appears in Figure 2.1.

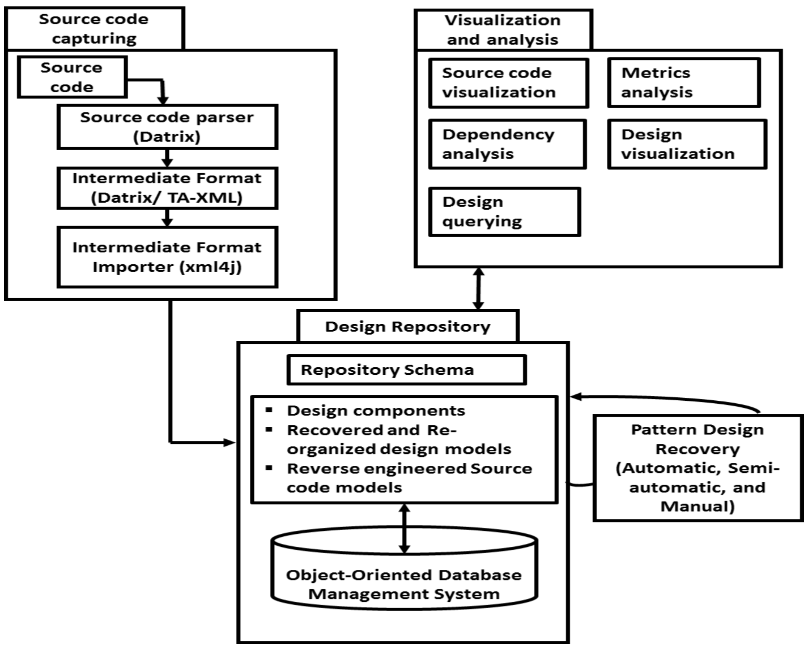

Source code capturing
Source code
Source code parser (Datrix)
Intermediate Format (Datrix/ TA-XML)
Intermediate Format Importer (xml4j)
Visualization and analysis
Source code visualization
Metrics analysis
Dependency analysis
Design visualization
Design querying
Design Repository
Repository Schema
▪ Design components
▪ Recovered and Re-organized design models
▪ Reverse engineered Source code models
Object-Oriented Database Management System
Pattern Design Recovery (Automatic, Semi-automatic, and Manual)

Figure 2.1: The SPOOL environment

SPOOL reads C++ files as input and uses a Datrix parser to parse them. Datrix represents the C++ source code in the form of an ASCII-based representation (Datrix/TA intermediate format). SPOOL converts Datrix TA files into XML syntax (Datrix/TA-XML). Moreover, SPOOL uses an XML parser (xml4j) to read the content of Datrix/TA-XML files.

SPOOL recovered different source code information, such as classes, structures, attributes, parameters, return types, call actions, object instantiations and friendship relations. The design pattern recovery process aims to structure parts of the class diagrams so that they resemble pattern diagrams. SPOOL supports manual and automatic design pattern recovery. Furthermore, SPOOL introduces the concept of a reference class (the class that most reflects the class behavior).

The SPOOL environment has been applied on three industrial systems. For confidentiality reasons, System A and System B were used to represent the first and second systems. The third system is ET++ v3.0. SPOOL recovered 43 instances of the Template Method pattern in system A (these instances of the Template Method design pattern were recovered by traversing all source code classes using the implementation of Template Method query, searching for local operations calls and verifying whether the "primitiveOperation" is polymorphic). Furthermore, SPOOL detected 46 bridge design instances in ET++ and the "Abstraction" participant class was selected as a reference class. Bridge query searches for classes with an instance variable of type "Implementor". In addition, it verifies whether the receiver of an operation call is of type "Implementor" and whether it is overridden by at least one subclass of "Implementor" (fingerprinting a bridge instance). The efficiency and accuracy of SPOOL were not reported.

2.2.2 Metrics-Based Approaches

Metrics-based approaches compute program related metrics, such as aggregations, associations and dependencies, from different source code representations. Different techniques are applied to compare pattern metric values to that of the source code. Metrics-based approaches reduce the search space through filtration.

2.2.2.1 MAISA

MAISA (Metrics for Analysis and Improvement of Software Architecture) is a research tool developed at the University of Helsinki [23]. MAISA represents design pattern detection as a constraint satisfaction problem (CSP) where problems are represented as a set of constraints over variables in a particular domain. A CSP involves a set V of variables, a domain Di for each variable i ∈ V and a set of constraints P. The target is to find an assignment S to the variables in such a way that the assignment satisfies all the constraints: $S = \{i: = x \mid p1 \wedge p2 \wedge \ldots \wedge pn \wedge p1j \wedge p2j \wedge \ldots \wedge pmj \; \forall\, pk \in Pi \; \forall pkj \in Pij \; \forall i, j \in V, x \in Di\}$.

MAISA represented CSP as a graph in which variables and their domains are represented as nodes. On the other hand, constraints are represented as arcs. An arc-consistency algorithm (AC-3) is used to delete the values that cannot satisfy certain constraints from the node domains. Values deletion is performed until a solution is found. Metric prediction attributes are stored in a library. A user can select the pattern that he wants to search for. MAISA will search for the selected pattern and provide each match as a potential candidate.

MAISA was implemented in Java. In addition, Prolog was used for the structural coding of software architecture and architectural patterns. Architecture and patterns are described as UML diagrams which will be translated later on to Prolog format. Pattern relationships and architecture components are expressed using Prolog facts. MAISA uses software metrics to measure software architecture and to estimate the final system. The architectural measurements comprise size and complexity metrics, such as the number of messages, the depth of the inheritance tree and the number of classes.

MAISA involves the following components: UML editor, pattern library, pattern miner, metric analyzer and reporting tool. MAISA have been applied to Nokia’s DX200 switching system and two instances of Abstract Factory design pattern were recovered. The efficiency and accuracy of MAISA were not reported.

2.2.2.2 FUJABA

FUJABA (From UML to Java and Back Again) is a design pattern detection tool in which design patterns are defined as sub-patterns [24]. The source code is represented as an Abstract Syntax Graph (ASG). Later, during the detection process, ASG is enriched with annotations that indicate the presence of sub-patterns in the source code.

FUJAPA applied transformation rules to capture the structural and behavioral aspects of design patterns. Transformation rules are organized into multiple levels of hierarchies. For example, level one of the hierarchy holds the source code information. FUJABA used a combined bottom up and top down strategy to apply the transformation rules. FUJAPA‘s detection algorithm uses assigned level numbers, which are associated with the transformation rules, to establish the orders of applying the rules on ASG. In addition, FUJABA uses fuzzy values to accept or reject the detected pattern elements (sub-patterns). The use of sub-patterns makes the detection process incremental. Hence, relevant information can be achieved in a short time. For example, FUJABA represented the Composite design pattern as three sub-patterns (Generalization, Association and Delegation). Generalization is recovered during level one, whereas Association and Delegation are recovered during level two.

FUJABA relies on class diagrams, activity diagrams and state chart representations of design patterns. Class diagrams describe design pattern structure, whereas activity diagrams and state chart diagrams describe the behavioral aspects of design patterns. One of the interesting features of FUJABA is its ability to generate codes from collaboration diagrams. The FUJABA environment has been applied to an automatic material transportation system. FUJABA is a semi-automatic tool which needs the intervention of a software engineer. FUJABA did not report any evaluation results to assess its effectiveness or efficiency.

2.2.2.3 Antoniol et al. Approach

The approach presented by Antoniol *et al.* [25] generates an Abstract Object Language (AOL) representation for both source code and subject system design. Class-level metrics, such as the number of aggregations, associations and inheritances, are computed as well. Specifically, a brute-force approach to identify all possible pattern candidates was adopted. To identify all pattern candidates in a design containing N classes, all possible arrangements of the classes and their relationships are computed.

The experiments have been performed on a public domain code and an industrial code in order to assess the approach effectiveness. The reported precision was 55%.

2.2.2.4 Detten and Becker Approach

The approach in [26] combines both clustering-based and pattern-based reverse engineering approaches. This approach shows that the occurrences of bad smells in the software system code can falsify the results of a metric-based clustering. Moreover, the approach applies pattern detection to an initial decomposition of the system to detect bad smells, thereby preventing the clustering algorithm performing a further decomposition.

2.2.2.5 Uchiyama Technique

The technique presented by Uchiyama *et al.* (hereafter, Uchiyama technique) uses source code metrics and machine learning to detect design patterns [27]. By using the goal question metric method (GQM), some source code metrics are selected to judge roles. Pattern specialists define a set of questions to be evaluated and select some metrics to help to answer these questions. Moreover, Uchiyama technique uses a hierarchical neural network simulator in which the input is metric measurements of each role and the output is the expected role. Figure 2.2 shows the technique presented by Uchiyama *et al.* [27].

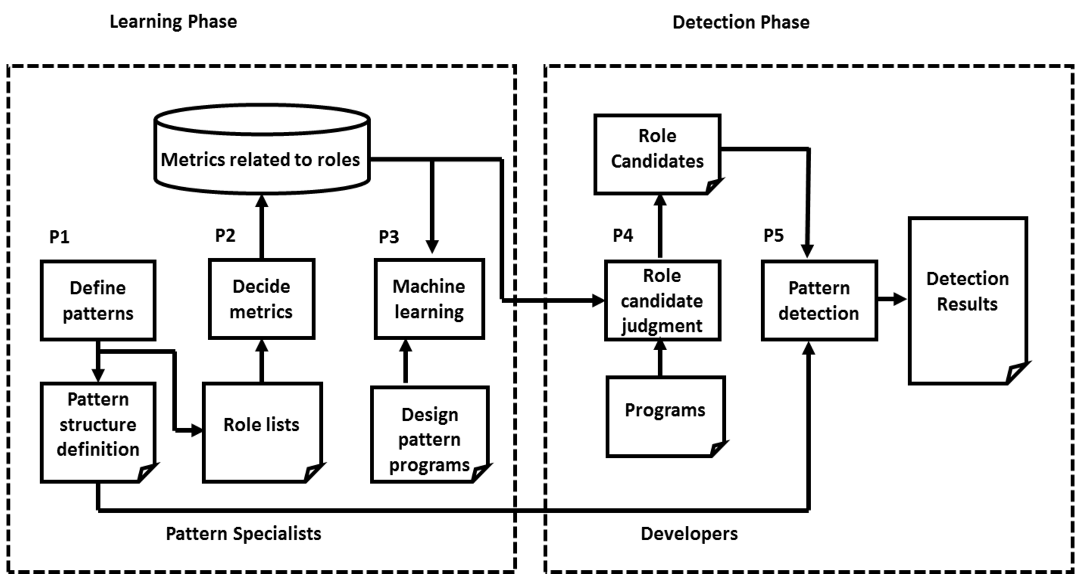


Figure 2.2: Uchiyama technique

The hierarchy number is set to three, the number of inputs is set to the number of decided metrics and the number of outputs is set to the number of roles. The sigmoid function has been used as a transfer function in the neural network. In addition, a back-propagation has been used to calculate the error margin between output Y and correct answer T. The Uchiyama technique detected five design patterns: Singleton, Adapter, Template method, State and Strategy.

The detection was done by matching the candidate roles, produced by the machine learning simulator, to the pattern structure definitions. Searching is looking for all possible combinations of candidate roles that are in agreement with pattern structures. The Uchiyama technique recovered inheritance, interface implementation and aggregation relationships. Pattern agreement values were calculated based on the role and relation agreement values. The Uchiyama technique has been applied to small-scale programs as well as to large-scale programs (Java library v1.6.0, JUnit v4.5 and Spring framework v2.5 RC2). A total of 40 pattern instances in the small-scale programs and 126 pattern instances in the large-scale programs were used as learning data. The Uchiyama technique has distinguished between the Strategy and State design patterns and is able to detect pattern variations. However, the validity of the parameters, expressions, agreement values and thresholds is not proved. The Uchiyama technique was evaluated in terms of recall and precision. The reported precision and recall rates were 63% and 76% respectively.

2.2.3 UML Structure, Graph and Matrix-Based Approaches

These approaches represent the structural and behavioral information of the subject system as UML structure, graph or matrix. Most of these approaches have good precision and recall rates but they are not capable of handling the implementation variants of design patterns.

2.2.3.1 Seemann and Gudenberg Approach

Seemann and Gudenberg, in their work, showed how to recover design information from the Java source code [28]. A compiler collects the relationships information, method calls and inheritance hierarchies and the result of the compile phase is a graph. A filtering was made to the graph to detect design patterns. Seemann and Gruenberg's approach detects only Strategy, Bridge and Composite design patterns.

2.2.3.2 DEPAIC++

DEPAIC++ (DEsign PAtterns Identification of C++ programs) is a design patterns detection tool developed by Espinoza *et al.* in 2002 [29]. DEPAIC++ is a canonical model formulated to analyze the structure of C++ classes. In addition, DEPAIC++ verifies whether or not the code being analyzed is using design patterns. DEPAIC++ is composed of two modules that first transform the C++ code into a canonical form and then recognize design patterns. However, DEPAIC++ did not analyze the behavior of the source program. It detects design patterns starting from a structural analysis of the source code, whereas some design patterns implement different behaviors in their solutions.

2.2.3.3 Columbus

Columbus is a reverse engineering framework developed at the University of Szeged to analyze C++ projects [30]. The recovered information is presented as Columbus schema for C++. The schema represents C++

elements at different levels of abstraction. The schema description is represented using a UML class diagram. Moreover, physical representations were created from a schema instance (Abstract Syntax Graph). The operation of Columbus is performed using three plug-ins:

- Extraction plug-in which analyzes the C++ source file and creates a file to store the recovered information. Columbus reads the input files and passes them to an extractor which will generate the appropriate internal representation. The extractor parses the input source file by invoking a separate program called CAN (C++ ANalyzer).
- Linker plug-in which builds a complete internal representation of the project. Columbus applied different filtering methods, such as filtering using C++ elements categories, filtering by input source files and filtering by scopes.
- Exporter plug-in which exports the internal representation to a given output format such as HTML, Graphic Exchange Language (GXL) and MAISA.

Columbus recovery capabilities were applied on three C++ projects: IBM Jikes Complier, Leda Graph Library and Star Office Writer. Columbus presented different statistics for the subject systems, such as number of classes, number of functions, memory consumption and CPU times. The accuracy of Columbus was not reported.

2.2.3.4 Similarity Scoring Approach

Design pattern detection using Similarity Scoring Approach (SSA) is a research prototype developed in Java at the University of Macedonia to handle the problem of multiple variants of design patterns [31]. SSA describes design patterns to be detected, as well as the subject system, as graphs. SSA represents all system static information as a set of matrices.

SSA uses a graph similarity algorithm to detect design patterns by calculating the similarity of vertices in the pattern and the subject system. To handle the system-size problem, SSA divides the system into a number of sub-systems and the similarity algorithm is applied to the sub-systems instead of the whole system. SSA was applied to three open source systems: JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7. Results were validated against the documentation of the systems.

Moreover, SSA uses matrices to represent the relationships between classes, which are directed graphs that can be mapped into a square matrix. To preserve the validity of the results, SSA similarity scores were bound within the range [0, 1]. SSA shows that the use of a similarity algorithm produces more accurate results than the use of exact/inexact graph matching.

SSA Methodology

Since design patterns involve class hierarchies, SSA locates these hierarchies and applies the similarity algorithm to them. The methodology presented by SSA can be summarized as follows:

- Each characteristic of the subject system (associations, generalization, abstract classes, object creations, abstract method invocations) is represented as a N×N adjacency matrix where N is the total number of classes in the subject system.
- Inheritance hierarchies are detected. Classes that do not participate in any hierarchy are listed in a separate group of classes.
- Building sub-system matrices. The whole system is partitioned into a number of sub-systems to improve detection efficiency. The experimental results show that the time required to apply the similarity algorithm to sub-systems is less than that for the whole system. Sub-systems are formed by merging all system hierarchies, two at a time.
- Appling the similarity algorithm between design pattern matrices and sub-system matrices by calculating normalized similarity scores.
- Recovering design patterns in each sub-system. A list is created for each pattern role. Sub-system similarity scores are sorted in descending order. Sub-system classes that have scores equal to the highest score of each role are inserted into the list. Exact matching, for

a pattern role, occurs when scores are equal to one. On the other hand, scores for modified pattern versions result in scores that are less than one.

To improve the accuracy of the detection process, SSA minimized the number of roles by considering only the important roles of each design pattern. In some cases, in which the sub-system does not involve the required pattern attributes, the detection process is terminated and the sub-system is excluded.

SSA Results Evaluation

SSA has been evaluated in terms of efficiency and accuracy. The validation of results was done manually by inspecting the source code and referring to the documentation of the subject systems.

In terms of accuracy, SSA obtained false positives for two patterns: Factory Method and State design patterns. SSA achieved a precision of 100% for all the examined design patterns and a recall ranging from 66.7% to 100%.

SSA detected the following design patterns: Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method and Visitor.

SSA efficiency was determined by measuring the CPU times and memory consumption during the detection process. SSA shows that the similarity algorithm is the most intensive task. Furthermore, storing the adjacency matrices is the main consumer of resources.

SSA has a number of limitations. For example, it assumes that no more than one characteristic for a given design pattern instance is modified. To distinguish true positives from false positives, SSA uses a threshold value. For the pattern roles that have two characteristics (association and generalization), a threshold value of 0.5 is assigned. Moreover, SSA cannot detect the characteristics that are external to sub-system boundary, such as chains of delegations. Finally, SSA does not employ any dynamic information.

2.2.3.5 DP-Miner

Design Patterns Discovery Matrix (DP-Miner) was developed at the University of Texas as a research prototype to detect design pattern instances [32]. DP-Miner represents system structure (all classes of the system) as a matrix of columns and rows. Each cell in the matrix represents a certain relationship between classes. On the other hand, the relationships between pattern participant classes are represented in another matrix. The detection of design patterns is performed by matching the two matrices. The architecture of DP-Miner appears in Figure 2.3.

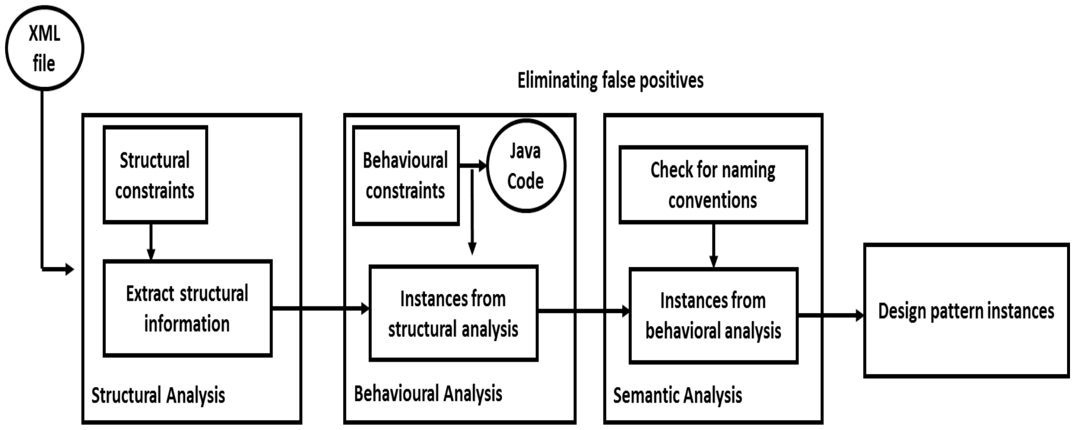

XML file
Eliminating false positives
Structural constraints
Extract structural information
Structural Analysis
Behavioural constraints
Java Code
Instances from structural analysis
Behavioural Analysis
Check for naming conventions
Instances from behavioral analysis
Semantic Analysis
Design pattern instances

Figure 2.3: DP-Miner architecture

DP-Miner used an IBM plugin called "UniSys" to transform UML class diagrams into XMI format. DP-Miner involves three analysis phases: structural, behavioral and semantic. In the structural analysis phase, DP-Miner takes the advantage of prime numbers where any given number can be broken into a unique list of prime numbers. A unique prime number is assigned to each structural element. "One" is excluded since it is used as an identity for multiplication. DP-Miner calculates the weight of each class according to the following formula:

Total class weight = $W_a \times W_m \times W_{as} \times W_g \times W_d \times W_{ag}$

Where: W_a: $2^{\text{number of attributes in the class}}$, W_m: $3^{\text{number of methods in the class}}$, W_{as}: $5^{\text{number of association relationships in the class}}$, W_g: $7^{\text{number of generalisation relationships in the class}}$, W_d: $11^{\text{number of dependency relationships in the class}}$ and W_{ag}: $13^{\text{number of aggregation relationships in the class}}$

For example, if a class weight is calculated as 5670, then it can be broken into 21 × 34 × 51 × 71, which indicates that this class has one attribute, one association and one generalization. Optimization has been applied to handle the overflow problem which may occur because of the large numbers of methods and attributes. DP-Miner encoded the system design into n × n matrix (where n is the total number of classes in the system). On the other hand, design pattern relationships are encoded into m × m matrix (where m is the total number of participant classes). Design pattern detection is done by matching the system design matrix to the design pattern matrix. Each cell in the system design matrix is initialized by 1.

For each cell in the system design matrix, if the relationship between class I and class J is:

- Association: then cell value is multiplied by 5.
- Generalization: then cell value is multiplied by 7.
- Dependency: then cell value is multiplied by 11.

DP-Miner excluded the aggregation relationships from the structural analysis phase since it is difficult to extract them from an XMI file. On the other hand, DP-Miner checks class types such as abstract, concrete, or interface. The behavioral analysis phase focuses on finding dependency relationships and method delegations. To handle the problem of implementation variants, DP-Miner constructs a Control Flow Graph (CFG). However, it is not clear how the behavioral aspects are recovered by DP-Miner. How is the XMI file parsed?

Furthermore, DP-Miner uses semantic analysis to get clues from named conventions of classes. However, not all classes in the system may be named with pattern-related information (i.e. some implementations may be developed without any pattern knowledge).

DP-Miner Results Optimization

DP-Miner was applied to the Java AWT package (Abstract Windowing Toolkit). The AWT package involves 346 files which contain 485 classes and

111 interfaces. When DP-Miner is applied to a large number of attributes and methods, an overflow is noticed in practice. To handle this, DP-Miner optimized the weight calculations and the matrix construction.

Depending on the observation of GoF pattern characteristics, DP-Miner concluded that there was no more than one attribute in each pattern participant class. In addition, the number of operations in any pattern participant class is less than five. Based on the observation that there is only one instance of each relationship, the maximum weight for a given class is 25 × 35 × 5 × 7 × 13 = 31352832. Hence, DP-Miner changes the way of scanning the XMI file so the matrix can be built in one iteration. The matrix construction time of JavaAWT is reduced to 15 seconds.

DP-Miner Detection Methodology

DP-Miner recovery relies mainly on calculating class weights and on the construction of a relationship matrix. The weight of each class provides an indication of the number of attributes and operations inside each class and its relationship with other classes. It must be noted that DP-Miner did not distinguish between the inheritance and realization relationships (i.e. DP-Miner considers that the relationships of class A extend class B and that class A implements class B as an inheritance relationship). Moreover, DP-Miner did not recover the aggregation and dependency relationships.

DP-Miner recovered Adapter, Bridge, Strategy and Composite design patterns from the Java AWT package. To explain how DP-Miner works, an example of the recovery of the Adapter design pattern from JavaAWT is presented. DP-Miner calculates a weight for each Adapter participant class. The calculated weights are: Target class = 3 (one method), Adapter = 105 (one method, one association and one generalization) and Adaptee class = 3 (one method). The relationship matrix is also constructed to describe the relationships among Adapter participant classes. Table 2.1 shows the relationship matrix of the Adapter design pattern.

Class	**Target**	**Adapter**	**Adaptee**
Target	1	1	1
Adapter	7	1	5
Adaptee	1	1	1

Table 2.1: DP-Miner's Adapter pattern matrix

Candidate instances of the Adapter design pattern are identified during the structural analysis phase. False positives are reduced later during the behavioral analysis phase. The semantic analysis phase is not applied since there are no ambiguity issues between the Adapter instances. DP-Miner searches for weights of a group of classes in Java AWT that are integral multiples of the Adapter participant classes. The Adapter matrix may not have the exact sub-matrix of the JavaAWT matrix. Each candidate instance in the Adapter matrix must satisfy the following rules:

- Target class should be abstract or interface and its weight should be a multiple of 3.
- Adapter class should be concrete and its weight should be a multiple of 105.
- Adaptee class should be concrete and its weight should be a multiple of 3.

During the behavioral analysis, DP-Miner will check whether there is any method that plays the role of the request method (the request method is defined in the Target class and re-implemented in the Adapter class). If an instance satisfies the first condition, then the tool will check if the request method in the Adapter class invokes the request method in the Adaptee class.

DP-Miner detected 21 Adapter instances within 2.44 seconds and 3, 76 and 65 instances of Composite, Strategy and Bridge respectively. Result validations were performed based on the manual tracing of the Java AWT package and on the documentation provided by Java AWT. The detection accuracy was not reported.

2.2.3.6 Sub-Patterns Approach

The approach presented by Dongjin *et al.* involves a sub-pattern representation for the 23 GoF design patterns, henceforth the sub-patterns approach [33]. The source code and predefined GoF design patterns are transformed into graphs with classes as nodes and relationships as edges. The instances of sub-patterns are identified by means of sub-graph discovery. The Joint classes have been used to merge the sub-pattern instances. Moreover, the behavioral characteristics of method invocations are compared with a predefined method signature template of GoF patterns to obtain final instances.

The sub-patterns approach introduces a structural feature model to represent GoF design patterns. The structural feature model recovered four main relationships: inheritance, aggregation, association and dependency. In fact, the sub-patterns approach defined 15 sub-patterns to represent GoF design patterns. A class-relationship directed graph has been used to represent the classes and their relationships.

In addition, the sub-patterns approach uses a class-relationship directed graph to represent system classes and the relationships among them. The directed graph has been defined as follows:

$V = \{V_0, V_1 \ldots V_n\}$ is the set of vertices that represents system classes.

$E = \{e\ (v_i, v_j)\ C\ V \times V$ is the set of directed edges that represents the relationships.

$W = E \rightarrow W_E$ is the function that assigns weights to edges.

The prime numbers 2, 3, 5 and 7 were used to represent the weights of the association, inheritance, aggregation and dependency respectively. If two classes have more than one relationship; the corresponding prime numbers are multiplied.

The sub-patterns approach works in four main steps:

- Modelling the source codes into a class-relationship directed graph in which the vertices represent the classes and the edges, while their weights represent the relationships among classes.
- Detecting the instances of sub-patterns by matching the sub-graphs of the class-relationship directed graph to that of the predefined sub-patterns.
- Merging the sub-pattern instances based on the joint classes and comparing them to the Structural Feature Model to produce the candidate instances.
- Analyzing method signatures to obtain final instances.

The sub-patterns approach used Enterprise Architect, a visual platform for modelling software systems, to parse the source codes and to produce class diagrams and XMI files.

The searching algorithm will search for the candidate vertices in the class-relationship directed graph of the subject system (GCDRs). For each vertex (vi) in the class-relationship directed graph of the sub-pattern (GCDRm), the vertices in GCDRs are selected if their outbound composite weights and inbound composite weights can be divided, with no remainder, by those of vi in GCDRm. The candidate vertices are combined to generate k-sub graphs. Further details on how to calculate the outbound composite weights and inbound composite weights are presented in [33].

The sub-patterns approach has been applied to nine open source systems and a Design Pattern Instances Detection tool has been developed to implement the methodology. Precision, recall and F-measure metrics were used to assess the detection accuracy. Moreover, the execution time for the instances recovery, structural analysis and behavioral analysis was calculated. As it was reported by the authors, the sub-patterns approach spent a longer time on method signatures analysis than on structural analysis. The validation of the results is performed manually and the repository of Perceron [34] has been used as a reference benchmark. The sub-pattern approach achieved precision which ranges from 68% to 100% and recall with ranges from 73% to 100%.

We noticed contradictions in the results presented by the sub-pattern approach. For example, the authors stated that Perceron was used as the main reference to validate the recovered instances, but Perceron did not involve JHotDraw and JavaAWT in its list of open source projects. Furthermore, the reported results for the Factory Method recovery in JUnit was not accurate where the sub-pattern approach reported two instances and Perceron reported only one instance. The calculated precision and recall for the Factory method pattern were both 100%. Another example is the detection of the Proxy design pattern in Hodouk. Perceron reported zero instances, while the sub-pattern approach recovered 13 instances. The recall was calculated as 100%.

2.2.4 Miscellaneous Approaches

These approaches do not fit under any of the previous categories. The following is a brief description of each approach in this category.

2.2.4.1 Kraemer and Prechelt Approach

One of the first approaches to detect design patterns was presented by Kraemer and Prechelt in 1996 [35]. They tried to improve software maintainability through the detection of design patterns directly from the C++ source code. Design patterns are represented as Prolog rules which are used to query a repository of C++ codes.

The detection process focused on five structural design patterns: Adapter, Bridge, Composite, Decorator and Proxy. The Kraemer and Prechelt approach was applied to four projects: NME, ACD, LEDA and zApp class library. The reported precision was 14-50%.

2.2.4.2 PTIDEJ

PTIDEJ (Pattern Traces Identification, Detection and Enhancement in Java) was developed at the University of Montreal using Java under the Eclipse platform and, since then, PTIDEJ has evolved into a complete reverse engineering tool. PTIDEJ comprises several identification algorithms for design patterns, micro patterns and idiom patterns [36][37].

The core of the PTIDEJ tool is the PADL Meta-model (Pattern and Abstract Level Description Language). PADL involves a parser to generate a model of the subject system and to build an Application Object Library (AOL). PTIDEJ comprises four key tools:

- Software Architectural Defects (SAD) which can automatically detect the architectural defects and correct them by applying refactoring techniques at the design level.
- Efficient Patterns Identification (EPI) which is used to detect design pattern occurrences in the source code. EPI uses constraint-based programming to find exact and approximate occurrences of design patterns.
- Dynamic Relational Adjacency Matrix (DRAM) which is mainly used to visualize the dynamic relationships between classes.
- ASPECTS which computes certain metrics based on the generated aspect-oriented abstraction.

The architecture of PTIDEJ appears in Figure 2.4

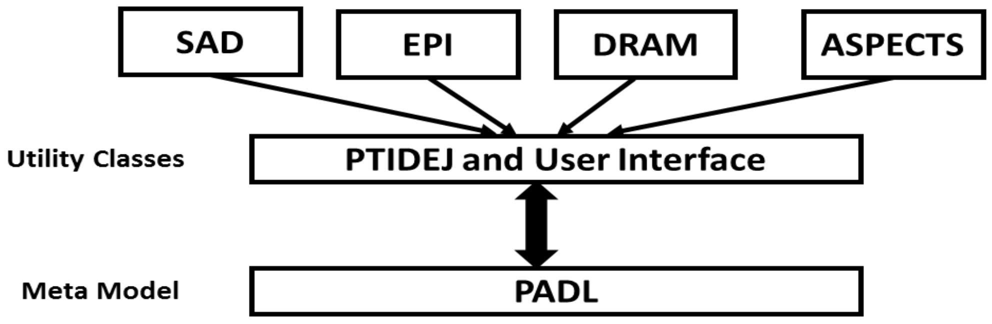


Figure 2.4: PTIDEJ architecture

Design solutions provided by design patterns are described in PTIDEJ using design motifs which are prototypical micro architectures. PTIDEJ recovered design patterns by finding all micro architectures that are similar to design motifs (i.e. finding all classes and interfaces that have structures similar to design motifs). Candidate design motifs are assessed qualitatively using quantitative signatures. In addition, the metric values of classes, which are playing roles in design motifs, are computed and sent to a rule learner algorithm.

Classes that are playing roles in design motif are evaluated using external class characteristics, such as size, filiation, cohesion and coupling. Since two or more classes may have identical external attributes and roles, fingerprinting cannot be used to detect design motifs. However, it can be used to reduce the search space for the candidate classes.

PTIDEJ considers design pattern detection as a constraint satisfaction problem (CSP) in which decisions were made during the variable assignment phase. Specifically, the explanation was used to explain the differences between the expected and the observed behavior for a given problem. In addition, the explanation was used to determine the constraint effects on the domain of variables. In the context of design pattern identification, explanation-based constraints can explain why no solution is found for a given problem. Moreover, relaxing a constraint allows the discovery of new solutions. The PTIDEJ detection process can be summarized as follows:

- Modelling design patterns as a constraint satisfaction problem (CSP). Design pattern participant classes are modelled as a set of constraints. An integer variable is associated with each participant class. Relationships between classes are represented as constraints over variables.
- Modelling the source code. Only the information needed to apply the constraints is kept, such as class names and relationships between classes.
- Finding the distorted solutions. When all real solutions of CSP are found, the user can guide the search process dynamically to find the distorted solutions. PTIDEJ has built a library of constraints to express the relationships between classes. The constraints library stores:

- Strict inheritance relationship. An inheritance relationship is expressed by "Strict Inheritance Constraint". If B inherits from A, then the constraint is expressed as A<B.
- Knowledge (if a method in class A invokes a method in class B). The knowledge is expressed by the constraint "RelatedClassesConstraint", expressed as A →B.
- Non-Knowledge (class A must know about class B). This relation is expressed by the constraint "UnRelatedClassesConstraint".
- Composition relationship (class A defines one or more fields of class B). This relation is expressed by the constraint "composition", expressed as A Ↄ B.
- Field type (to ensure that a field f of class A is of type B). This relation is expressed by the constraint "PropertyTypeConstraint", expressed by as A.f=B.

For example, PTIDEJ modelled the Composite design pattern by associating each participant class with a variable. Variable values are constructed based on the relationships between classes. Specifically, composite < component, leaf < component and composite Ↄ component. PTIDEJ used the explanation-based constraint solver, PALM, to extract a similar set of classes from the source code.

PTIDEJ has been applied on two packages of the Java class libraries: Java AWT and Java.NET. As reported by PTIDEJ, all Composite and Façade design instances were identified correctly. However, the accuracy of PTIDEJ is not reported.

2.2.4.3 CrocoPat

CrocoPat is a tool for design pattern detection developed at the Technical University of Cottbus [38]. It represents the software metamodel in terms of relations. Design patterns are described by relational expressions. The main motivation for building CrocoPat is to handle the performance problem of the previous detection tools. The metamodel presented by CrocoPat divides the object-oriented program into packages, classes, methods and attributes.

CrocoPat automatically analyzes the object-oriented program and the user is able to define new patterns. Moreover, CrocoPat is able to analyze large, object-oriented programs in an acceptable time. In terms of graph theory, CrocoPat does sub-graph search. In terms of relational algebra, CrocoPat searches for tuples fulfilling a given predictive expression. CrocoPat represented all system relationships using a Binary Decision Diagram (BDD).

CrocoPat Detection Methodology

CrocoPat recovers design pattern instances using three main steps:

- Recovering source code data using a program analysis tool (sotograp). The recovered data will be stored in a relation file.
- Creation of pattern definitions using pattern specification language. The CrocoPat's language uses relational algebra expressions to express the pattern definitions. The syntax and semantics of the expressions are also defined. Specifically, CrocoPat defines U (Universe) as a set of all values and X as a finite set of all attributes. A tuple t of X is a total function t: X→ U. Val (X) is the set of all tuples of X.
- Recovering of the call, inherit and contain relationships.

CrocoPat Evaluation

CrocoPat has been applied on three open source systems: Mozilla, JWAM, and wxWindows. For example, to describe the Composite design pattern, CrocoPat used the expression Call Λ Inherits [x/y] Λ contains [x/z] Λ (inherits [x/L] Λ ^{r} (contains [x/L]) where x is the client class, y is the component class, z is the composite class and L is the leaf class. Moreover, CrocoPat recovered some design analysis structures, such as classes in circles (class x is used by other classes including itself) and the role of identity (two classes use the same classes).

CrocoPat is only evaluated in terms of performance. The reported results only show the detection of the Composite and Mediator instances in Mozilla, JWAM and wxWindows. The recovery performance for the detection of all Composite instances in the three subject systems was 23 seconds for Mozilla, 3.1 seconds for JWAM and 1.0 seconds for wxWindows.

2.2.4.4 SPQR

System for Pattern Query and Recognition (SPQR) is a tool-set for elemental design pattern detection in C++ source code, developed at the University of Carolina [39]. SPQR uses a logical inference system to encode rules, which will be combined later to form patterns using reliance operators, and to encode the structural/behavioral relationships between classes and objects using rho-calculus. SPQR components are presented in Figure 2.5.

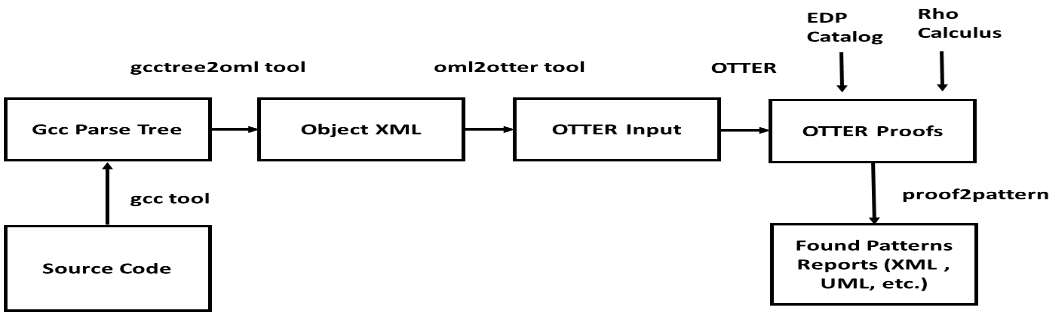


Figure 2.5: SPQR components

SPQR analyses C++ source code for a particular syntactic structure which matches the P-calculus concepts. SPQR uses a “gcc” tool to generate an abstract syntax tree representation of the source code. A “gcctree2oml” tool has been developed by SPQR to read the abstract syntax tree and to generate an XML representation of the object structure. Furthermore, the “oml2otter” tool reads the object’s XML file and produces a feature rule file (otter input) which will be used by the automated theorem prover (OTTER) to find design pattern instances. Finally, a “proof2pattern” tool analyses the OTTER proof and produces a final pattern report.

SPQR is only applied to Killer Widget Application and the Decorator design pattern is recovered. SPQR results were only validated in terms of efficiency (CPU times and memory consumption).

2.2.4.5 PINOT

Pattern Inference and Recovery Tool (PINOT) reclassifies the catalog of design patterns by intent [40]. PINOT was built from Jikes, an open source Java compiler, and focuses on the detection of common design patterns used in practice. To capture program intent, PINOT used static program analysis techniques to recover design pattern instances from four open source projects: Java AWT v1.3, JHotDraw v6.0, Java Swing v1.4 and Apache Ant v1.6. PINOT reclassifies GoF design patterns, based on their structural and behavioral similarities, into five groups: Language-provided patterns, Structural driven patterns, Behavioral driven patterns, Domain-specific patterns and Generic concepts.

Language-driven patterns concern the patterns already implemented by programming languages, such as the Iterator and the Prototype design patterns, where they are implemented in Java using the libraries Java.util.Iterator and Java.lang.Object respectively. Structural driven patterns describe the inter-class relationships (generalization, association and delegation). These patterns are Bridge, Composite, Adapter, Façade, Proxy, Template method and Visitor. Behavioral driven patterns encode all pattern behaviors inside their method bodies. These patterns are Singleton, Abstract Factory, Factory method, Flyweight, Chain of responsibility, Decorator, Strategy, State, Observer and Mediator. Interpreter and Command design patterns are classified as Domain-specific patterns. PINOT claims that the Interpreter design pattern uses the structure of the Composite design pattern and the behavior of the Visitor design pattern. Generic concept patterns involve Memento and Builder design patterns since they have a lack of structural and behavioral aspects. PINOT focuses on the detection of structural and behavioral patterns. Language provided, Domain-specific and Generic concepts patterns are excluded from the detection process.

Structural driven patterns are detected based on the relationships between classes. In addition, the virtual delegations, call dependencies, context interfaces, associations, aggregations, Factory interfaces and

Singleton class structures are identified. PINOT used data flow analysis on Abstract Syntax Trees (ASTs), in terms of blocks, to detect the behavioral driven patterns. Method bodies are represented as a Control Flow Graph (CFG) which is scanned later to determine method behaviors. Figure 2.6 shows the CFG used by PINOT to represent the “getInstance” method, used to determine whether a class has only one instance (the intent of Singleton design pattern) [40].

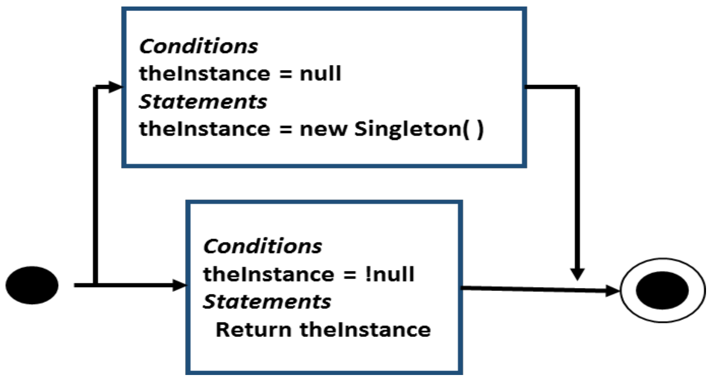


Figure 2.6: CFG of getInstance

PINOT was tested against the demo pattern source codes from the applied Java patterns. PINOT defined a pattern instance as a set of pattern participants’ classes. PINOT successfully detected the following patterns: Abstract Factory, Factory Method, Singleton, Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy, Chain of Responsibility, Mediator, Observer, State, Strategy, Template Method and Visitor.

PINOT has been run on a Linux machine using a 3GHz Intel processor with 1 G RAM. PINOT reported only the CPU times required to detect the structural and behavioral driven patterns in four open source projects (ANT, AWT, JHotDraw and Swing). CPU times for each subject system are presented in Table 2.1. PINOT results were validated against an authoritative discussion board and a developer documentation and also manually.

Subject system	**CPU Times (seconds)**
Ant	12.52
AWT	10.68
JHotDraw	8.98
Swing	66.79

Table 2.2: CPU times of PINOT's subject systems

2.2.4.6 DeMIMA

DeMIMA (Design Motif Identification: Multilayered Approach) is a semi-automatic tool, developed at the University of Montreal, that identifies micro architectures similar to design motifs in the source code [41]. DeMIMA involves three layers: two layers to generate the source code abstract model and class relationships and one layer to recognize design patterns from the generated abstract model.

DeMIMA uses the term "design motif identification" to describe the detection process. In addition, it defines microarchitecture as a set of classes, methods and relationships having a structure similar to one or more design motifs (patterns).

The DeMIMA detection process can be summarized as follows:

- Identifying micro architecture (mA) similar to a set of pattern motifs. Source code (S) will be analyzed to search for a set of design motifs.
- Contextualizing microarchitecture. This is done by storing design pattern motifs using external data.
- Understanding source code by representing it as a class diagram to describe design motifs.

DeMIMA provides an automation tool to implement the first task. Since the second and third task rely on human experience and system domain, they are not implemented. DeMIMA uses three layers to identify design motifs in source code: source code model, idiom level model and design level model. The traceability link between layers from source code up to the identified micro architecture appears in Figure 2.7. Moreover, DeMIMA defines a Meta-model, PADL (Pattern and Abstract Level Description Language), to express the characteristics of the class diagram.

$$\begin{tikzcd} S \arrow[r, shift left] & M_s \arrow[l, shift left] \arrow[r, shift left] & M_I \arrow[l, shift left] \arrow[r, shift left] & M_D \arrow[l, shift left] \end{tikzcd}$$

Where:
S: Source Code
M_s: Source Code Model
M_I: Idiom Model
M_D: Design Model

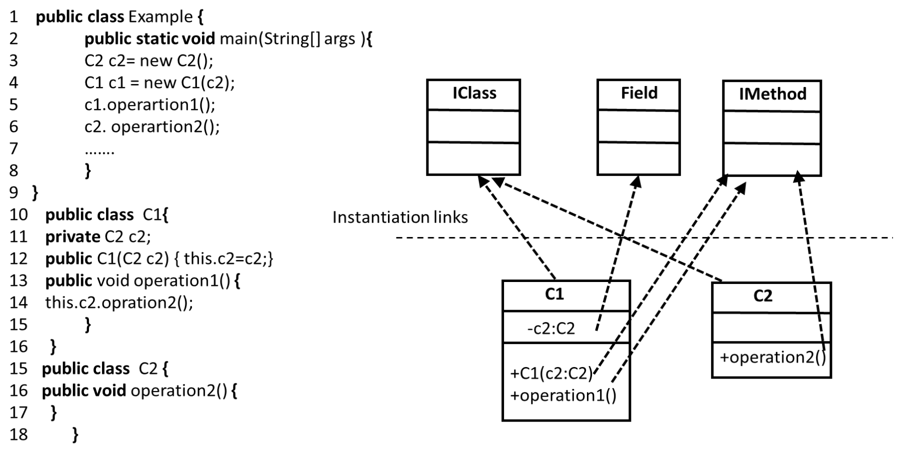
Figure 2.7: DeMIMA tractability link between layers

DeMIMA First Layer (Source Code Model)

DeMIMA first layer uses a parser to model the subject source code. The first layer includes all information found directly in Java source code, such as classes, methods and relationships. DeMIMA provides a Meta-model to describe the source code in terms of two parts:

- A class entity which describes the system as a set of classes.
- The class element which describes the methods inside each class.

A UML-like diagram is used to describe the source code model. A code example and its UML representation, as presented by DeMIMA, appears in Figure 2.8 [41]:



```
public class Example {
        public static void main(String[] args ){
        C2 c2= new C2();
        C1 c1 = new C1(c2);
        c1.operartion1();
        c2. operartion2();
        .......
        }
}
 public class  C1{
 private C2 c2;
 public C1(C2 c2) { this.c2=c2;}
 public void operation1() {
 this.c2.opration2();
        }
 }
public class  C2 {
public void operation2() {
 }
    }
```

Figure 2.8: DeMIMA source code model example

DeMIMA Second Layer (Idiom Level Model)

DeMIMA second layer provides a high-level abstraction view of the subject source code. Idioms specify the binary relationships between classes. DeMIMA focuses on the use, association, aggregation and composition of unidirectional class relationships. Four language-independent properties were used to define each relationship:

- Exclusivity property (EX): An instance of class B involved at a given time with an instance of class A may also participate in other relationships at the same time.
- Type of Receiver (RT): class A instances involved in a relationship send a message to class B instances. Message receivers can be fields, parameters, or local variables.
- Lifetime property (LT) which constrains the lifetime for all class B instances with respect to the lifetime of class A instances. LT relates the time of destruction between two instances of class A and class B. In object-oriented programming, garbage collection is used to control instances' lifetime.
- Multiplicity property (MU) which is the number of instances of class B allowed in a relationship.

Based on the above properties, DeMIMA formalizes the aggregation, composition and association relationships. Exclusivity and receiver type are considered static properties. In contrast, lifetime and multiplicity are considered dynamic properties. DeMIMA recovered the following relationships from the source code of Figure 2.8: Association (c1, c2) = False, Aggregation (c1, c2) = True and Composition (c1, c2) = False.

DeMIMA Third Layer (Design Level Model)

DeMIMA layer three models the design motifs as a set of participant classes. For example, Figure 2.9 shows a UML-like diagram representation of the source code of Figure 2.8.

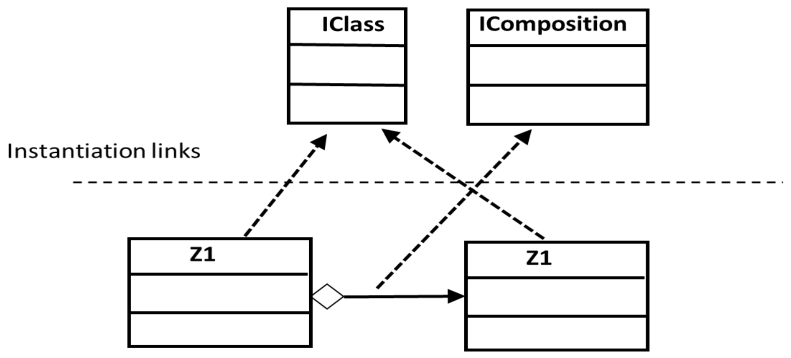


Figure 2.9: DeMIMA design motif of the source code example

DeMIMA uses explanation-based constraint programming to handle the constraint satisfaction problem. DeMIMA identifies micro-architectures similar to the design motifs by transforming them into constraints that reflect the relationships between the pattern's participant classes. The used constraints are inheritance constraint, strict transitive inheritance constraint, transitive inheritance constraint, use constraint, ignorance constraint and creation constraint. For each constraint, a weight is assigned, which is an integer value ranging from 1 to 100, to reflect the importance of the constraint. DeMIMA uses constraint relaxation to replace the constraints that lead to conflicts with semantically weaker constraints.

DeMIMA Implementation

DeMIMA was implemented using Java programming language on the top of the PTIDEJ framework [36][37]. DeMIMA uses the following PTIDEJ components:

- PADL (Pattern and Abstraction Level Description Language) to describe the models of source code, idiom and design.
- PADL class file creator to parse the Java source code files.
- Relationship static analyzer, used to compute the receiver type values and multiplicity values.
- CAFFINE, used to perform the dynamic analysis of the subject system by calculating lifetime and exclusivity.

- PTIDEJ user interface, used to visualize the model.
- PTIDEJ solver, used to produce microarchitectures.

Furthermore, DeMIMA detected twelve design motifs: Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method and Visitor.

Recall and precision were used to assess the effectiveness of DeMIMA. The experiments have been applied on JHotDraw v5.1, JRefactory v2.6.34, JUnit v3.7, MapperXML v1.9.7, QuickUML v2001 and 33 components. DeMIMA observed precision of 34% for the 12 design motifs considered and achieved 100% recall for the five open source systems. Due to confidentiality reasons, precision and recall were not reported for the industrial components.

2.2.4.7 DPRE

Design Patterns Recovery Environment (DPRE) is a design pattern recovery prototype developed at the University of Salerno [42]. DPRE uses a two-phase approach to recover structural design patterns from object-oriented code. Figure 2.10 shows the DPRE recovery process. DPRE phase one provides a coarse-grained level where design pattern candidates are identified by analyzing class diagram information recovered during the preliminary analysis. Class diagram information, such as name, type, inheritance relationships and association relationships, are parsed using a visual language technique. All information is stored in a repository for later use during the structural analysis phase. Method invocations and declarations are also stored.

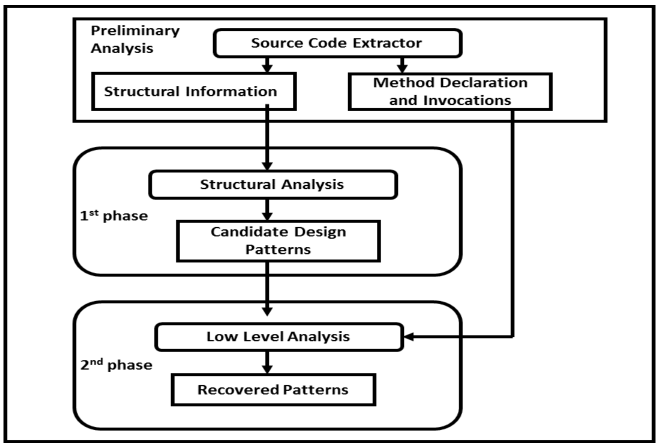


Figure 2.10: DPRE recovery process

In the second phase, codes of classes that participate in design pattern identification were examined to check their compliance with the corresponding GoF patterns' source code. The DPRE visual parser analyses the attribute-based representation of the pattern's class diagram, which is represented as a textual sentence to describe its classes and their relationships. DPRE representation of the Adapter design pattern appears in Figure 2.11 [42]. The generated textual representation of the Adapter design pattern is [CLASS Link$_{1,2}$ inheritance Link $_{1,1}$ CLASS' Link $_{1,1}$ Association Link $_{2,1}$class"].

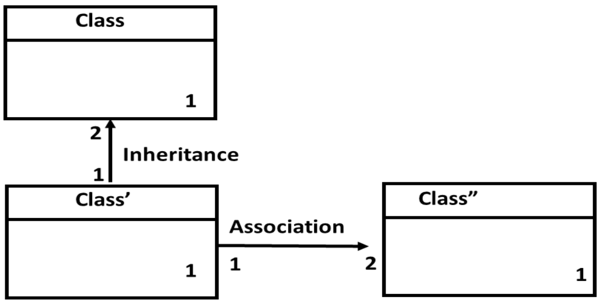


Figure 2.11: DPRE representation of the Adapter design pattern

DPRE examined six open source Java projects: JHotDraw v5.1, Apache Ant v1.6.2, JHotDraw v6.0b1, QuickUML 2001, Swing and Eclipse JDT components (Core v3.3.3 and User Interface v3.3.2). The effectiveness of DPRE is characterized by precision ranging from 62% to 97%. However, disparity in the results for the detected pattern instances is noticed.

2.2.4.8 MARPLE

Zanoni introduced an Eclipse plug-in called MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse) which supports the detection of both design pattern instances and software architecture reconstruction activities [43]. MARPLE tries to handle the variant problems of design pattern detection through the detection of sub-components called “basic elements”. The architecture of MARPLE involves five main modules that interact with each other through XML data transfer. These modules are:

- The Information Detector Engine module which uses the AST representation of the subject system to collect basic class elements and metrics.
- The Joiner module which recovers all design pattern candidates. Furthermore, it also represents the subject system as an Attributed Relational Graph (ARG). ARG has a set of vertices that correspond to a set of types and a set of edges that correspond to a set of basic elements which connects the types with each other.
- The classifier which tries to detect the possible false positives identified by the Joiner.
- The Software Architecture Reconstruction module which obtains abstractions from the subject system based on the metrics and the basic elements recovered by the Information Detector Engine.
- The Output Generation module which views the system analysis results.

The Information Detector Engine tries to extract architectures that match the target structure defined in terms of Joiner rules. The basic elements were recovered by visitors that parse the AST representation of the subject system source code. The instances of the basic elements that have been found inside a subject system are stored in an XML file. MARPLE constructs all the possible valid mappings, {(R_1, C_1), (R_2, C_2)… (R_n, C_n)}, for each pattern instance (C_i is a class that is supposed to play a role R_i inside the pattern).

2.2.4.9 Sempatrec

The approach presented by Alnusair *et al* [44] - henceforth Sempatrec - uses ontology formalism to represent the conceptual knowledge of the source code and semantic rules to capture the structure and behavior of design patterns.

A tool named Sempatrec (SEMantic PATtern RECovery) has been developed as a plug-in for the Eclipse IDE to implement the approach. Sempatrec processes the Java bytecode of the subject system, generates an RDF (Resource Description Framework) ontology and stores the ontology locally in a pool.

Specifically, Sempatrec generates a Source Code Representation Ontology (SCRO) to provide an explicit representation of the conceptual knowledge structure found in the source code. In addition, the developed SCRO serves as a basis for design pattern recovery where a design pattern ontology sub-model will be created. This sub-model extends the SCRO's vocabularies and involves an upper design pattern ontology that is further extended with a specific ontology for each design pattern.

Sempatrec utilized the Web Ontology Language (OWL), Resource Description Framework (RDF), Semantic Web Rule Language (SWRL), SPARQL protocol and a query language. OWL-DL is a sub-language of OWL which is based on the Description Logic (DL). More specifically, Sempatrec used OWL-DL and RDF to obtain a precise formal representation of various source code artifacts.

The Sempatrec recovery process focuses on four main relationships: aggregation, use, inheritance and realization. The object property "hasPart" and its inverse "isPartof" were used to represent the aggregation relationship. The "hasPart" property, which is a sub-property of the use object property, describes the "use" relationship.

Furthermore, Sempatrec has developed a knowledge generator sub-system that automatically extracts knowledge from Java bytecode. The proposed sub-system performs a comprehensive parsing of the Java class files and captures every SCRO element. In addition, the sub-system generates a set of instances for all ontological properties defined in SCRO.

The semantic instances, generated by the knowledge generator sub-system, are serialized using RDF and linked to SCRO or any other OWL ontology via OWL re-use mechanisms. In addition, the knowledge generator parses the subject system and extracts its structural descriptions. These descriptions are used to generate a separate RDF ontology which is represented using the Notation3 (N3).

To detect design patterns, Sempatrec defines a set of SCRO and SWRL rules to describe the pattern's structure and behavior. Moreover, an OWL-DL reasoner computes the entailments from a set of facts and SWRL rules defined in the ontologies. The detection process relies on a logical inference engine which requires a rule-based reasoner capable of processing the SWRL rules. After processing the rules, the reasoner will recover pattern instances by matching semantic constraints, specified in the rules, to the source code descriptions found in the knowledge base that represents the subject system.

SWRL rules have been written for 11 design patterns: Singleton, Factory Method, Abstract Factory, Composite, Adapter, Decorator, Template Method, Observer, Visitor and State/Strategy. Sempatrec automatically loads the required ontologies, calls the reasoner, executes the rules and runs built-in SPARQL queries to capture and display the recovered pattern instances.

Sempatrec has been applied to three open source systems: JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7. Precision and recall were used to assess the accuracy of Sempatrec which achieved an average of 82% and 90% for precision and recall respectively for the detection of Singleton, Factory Method, Abstract Factory, Composite, Adapter and Decorator design patterns. For the detection of the behavioral patterns, Template Method, Observer, Visitor and State/Strategy, Sempatrec achieved an average precision of 61% and an average recall of 88%.

Moreover, Sempatrec spent three seconds, four seconds, and 13.5 seconds parsing the framework, processing the ontologies and preparing the knowledge base from JUnit, JHotDraw and JRefactory respectively. On the other hand, the runtime that the reasoner spent executing the SWRL rules for all patterns was 28 seconds for JUnit, 3.6 minutes for JHotDraw, and 11.3 minutes for JRefactory.

In fact, it is not clear how Sempatrec formed the design pattern instances, although some instances are formed based on a certain role in the pattern. For example, a decorator instance only represents the decorator class and the roles of "Component", "ConcreteComponent" and "ConcreteDecorator" are ignored. Other disadvantages of Sempatrec can be noticed, such as its inability to establish the scalability of the reasoner's performance and the possible variations of the SWRL rules which may affect the precision and recall and the runtime performance. In addition, the built-in rules can only be edited using a specialized ontology editor. The main advantage of Sempatrec is the use of a pure ontology-based knowledge representation mechanism which ensures a consistent and formal functional representation of design patterns. Sempatrec is precise, practical and extensible.

Table 2.3 summarizes the whole spectrum of design pattern detection approaches. Some of the miscellaneous approaches are listed in the table and do not appear in this section (Pat [35], KT [45], DP++ [46], Kim and Boldyreff [47], Heuzeroth *et al.* [48], Philippow *et al.* [49], HEDGEHOG [50], and Kaczor *et al.* [51]). However, ALL table approaches are involved in the statistical analysis of this section. In addition, Table 2.3 presents the analysis style conducted by each detection approach where pattern detection approaches are classified into structural analysis approaches, behavioral analysis approaches and semantic analysis approaches.

Structural analysis approaches detect the instances of design patterns based on the static parts of the subject system. They explore inter-class relationships, method invocations and data types.

Behavioral analysis approaches consider the execution behavior of the program. The behavioral aspects of a program are recovered using static and dynamic analysis techniques. Behavioral analysis is useful since the structure of design patterns is not enough to provide a fingerprint inside the source code. For example, State and Strategy patterns have similar structures. Similarly, Chain of Responsibility, Proxy and Decorator patterns have identical structures. However, the possible variants of the same implemented behavior can increase the number of false positive instances.

Semantic analysis complements the structural and behavioral aspects to reduce the number of false positive instances. Naming conventions and annotations were used to retrieve the role information. Semantic information is important to distinguish between design patterns that have identical structural and behavioral aspects, such as State, Strategy and Bridge.

Detection Methodology	Tool/ Author	Year	Analysis Style	R
Database Query Approaches	Rasool *et al.*	2010	ST, SE	[18]
	D3	2008	ST, BE	[19]
	Marek Vokac	2006	ST	[20]
	SPOOL	1999	ST	[22]
Metrics-Based Approaches	MAISA	2000	ST	[23]
	FUJAPA	2002	ST,BE	[24]
	Antoniol *et al.*	1998	ST,BE	[25]
	Detten and Becker	2011	ST	[26]
	Uchiyama *et al.*	2014	ST,BE	[27]
UML Structure, Graph and Matrix Based Approaches	Seemann and Gudenberg	1998	ST,SE	[28]
	DEPAIC++	2002	ST	[29]
	Columbus	2002	ST	[30]
	SSA	2006	ST	[31]
	DP-Miner	2007	ST,BE,SE	[32]
	Dongjin *et al.*	2015	ST,BE	[33]
Miscellaneous Approaches	Pat	1996	ST	[35]
	PTIDEJ	2001 2004	ST	[36][37]
	CrocoPat	2003	ST	[38]
	SPQR	2003	ST	[39]
	PINOT	2006	ST,BE	[40]
	DeMIMA	2008	ST	[41]
	DPRE	2009	ST	[42]
	MARPLE	2012	ST,BE	[43]
	Sempatrec	2014	ST,SE	[44]
	KT	1996	ST	[45]
	DP++	1998	ST	[46]
	Kim and Boldyreff	2000	ST	[47]
	Heuzeroth *et al.*	2003	ST,BE	[48]
	Philippow *et al.*	2005	ST	[49]
	HEDGEHOG	2005	ST,BE,SE	[50]
	Kaczor *et al.*	2006	ST	[51]
Note: **ST:** Structural Analysis **BE:** Behavioral Analysis **SE:** Semantic Analysis **R:** Reference				

Table 2.3: Summary of detection approaches based on their detection methodology and analysis style

2.3 Analysis and Discussion

Design patterns are flexible design templates that may have several implementations. However, design patterns are described informally, which may cause misunderstanding. With the trend of applying new technologies, new approaches and tools are continuously being proposed. This section aims to provide a comprehensive comparison between all design pattern detection approaches in terms of subject system representation, subject systems, recovered design patterns and evaluation criteria.

2.3.1 Intermediate Representation of the Source Code

To the best of our knowledge, all detection approaches in the literature are targeting the source code of the subject system and avoiding targeting the system's design model to extract the instances of design patterns. The design model does not provide any runtime data necessary for the design patterns' recovery (for example, the association relationships). Usually, the design documents are inconsistent with the source code. Furthermore, most of the design models are not publicly available. All these reasons made the source code a better choice than the design model for recovering the instances of design patterns.

Most design pattern detection approaches use Abstract Syntax Tree (AST) representation to generate a source code model. The source code model should hold all the required information to recover design pattern instances. Table 2.4 lists the intermediate representation used by different detection approaches.

Some approaches used their own defined representation, such as [36], [41] and [51]. These approaches defined PADL, Pattern and Abstract Level Description Language to recover the source code information. Two approaches did not generate an intermediate representation of the source code, [43] and [47]. Rather, these approaches used software metrics to gather source code information. However, each detection approach may use a certain representation in a different format. For example, DPRE [42] uses

AST representation to generate a graph to represent class diagrams of a subject system. On the other hand, Heuzeroth *et al.* [48] use AST representation to define the static aspects of the patterns and the Temporal Logic Actions (TLA) to represent their dynamic aspects.

System Representation	**Author(s)/Tool**
AST (Abstract Syntax Tree)	Antoniol *et al.* [25], Detten and Becker [26], PINOT [40], DPRE [42], MARPLE [43], KT [45], Heuzeroth *et al.* [48], HEDGEHOG [50]
ASG (Abstract Syntax Graph)	FUJABA [24] Columbus [30]
UML, Graph	SPOOL [22], Seemann and Gudenberg [28], Dongjin *et al.* [33], DP++ [46], Philippow *et al.* [49].
Matrix	SSA [31] DP-Miner [32]
Prolog	MAISA [23] Pat [35]
PADL	PTIDEJ [36][37], DeMIMA [41], Kaczor *et al.* [51]
Metadata	D3 [19] Marek Vokac [20]
Other representations	Canonical form (DEPAIC++ [29]) Annotations (Rasool *et al.* [18]) BDDs (CrocoPat [38]) OTTER (SPQR [39]) SCRO (Sempatrec [44])
No representation	Uchiyama *et al.* [27], Kim and Boldyreff [47]

Table 2.4: The intermediate representation used by existing approaches

2.3.2 Subject Systems

The majority of detection approaches targeted open source codes that have been programmed using Java or C++. Two approaches, MAISA [23] and DP-Miner [32], targeted UML and XML open source systems. KT [45] applied its detection methodology to Smalltalk programs. Only one approach, CrocoPat [38], conducted its experiments on both Java and C++ open source systems. Figure 2.12 shows the programming languages used to program the subject systems. In fact, most of the detection approaches that have been introduced after 2008 applied their experiments to Java open source programs.

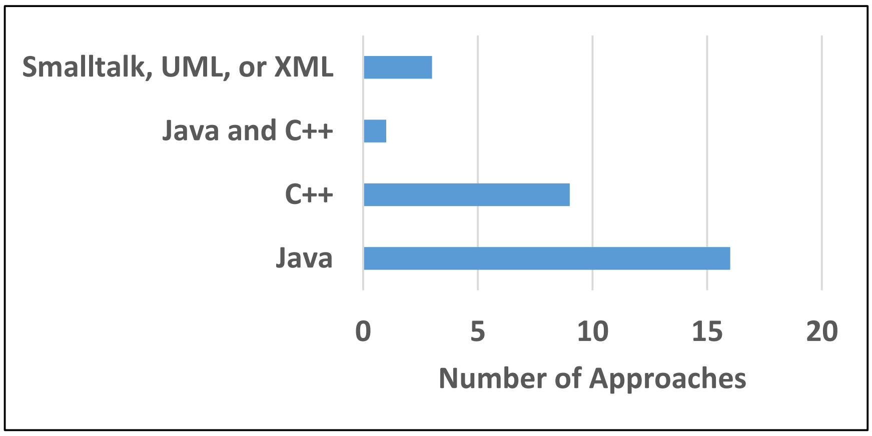


Figure 2.12: Programming languages used to program the subject systems

Furthermore, the detection approaches used different open source systems to evaluate their methodologies. The most commonly used open source systems are JHotDraw v5.1, JRefactory v2.6.24, JUnit v3.7 and QuickUML 2001. The selection of these approaches was made because:

- They used some well-known design patterns.
- The authors and the relevant literature indicate explicitly the implemented design patterns in the documentation.
- They are open source and their codes are publicly available.
- They vary in size.

Table 2.5 lists the subject systems used by different detection approaches to evaluate their detection methodology. It is clear that there is no common agreement in the literature on the appropriate subject systems for evaluating any new detection approach. In addition, the number of required subject systems is not clear. For example, some approaches apply their experiments to more than five subject systems while other approaches only apply their experiments to two subject systems. DeMIMA [41] applied its methodology to 33 industrial components, but there is no information about them.

Tool/ Author	**Subject systems**
Rasool *et al.* [18]	JHotDraw v6.1.2 and Apache Ant v1.6.2
D3 [19]	Applied Java Patterns and JHotDraw v6.0.b1
Marek Vokac [20]	Customer Relationship Management system
SPOOL [22]	ET++ and two telecommunication systems
MAISA [23]	Nokia DX200 switching system
FUJAPA [24]	Java AWT
Antoniol *et al.* [25]	LEDA, Libg++, Galib, Mec, Socket and 8 small-size industrial systems
Detten and Becker [26]	Common Component Modelling Example
Uchiyama *et al.* [27]	Java library v1.6.0, JUnit v4.5 and Spring v2.5
Seemann and Gudenberg [28], DEPAIC++ [29]	Not mentioned
Columbus [30]	IBM Jikes compiler, LEDA graph library and Star office writer
SSA [31]	JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7
DP-Miner [32]	Java AWT
Dongjin *et al.* [33]	Java AWT v5.0, JHotDraw v5.1, JUnit v3.8, Dom4J v1.6.1, Lizzy v1.1.1, Hodoku v2.1.1, Barcode4j v2.1.0, RstpProxy v3.0 and Teamcenter
Pat [35]	NME, LEDA and zApp
PTIDEJ [36][37]	Java AWT, Java.net packages, JHotDraw v5.1, JRefactory v2.6.24, JUnit v3.7, Lexi v0.0.1α, Netbeans v1.0.x and QuickUML 2001
CrocoPat [38]	Mozilla, JWAM and wxWindows
SPQR [39]	Killer Widget Application
PINOT [40]	Java AWT v1.3, JHotDraw v6.0, Java Swing v1.4 and Apache Ant v1.6
DeMIMA [41]	JHotDraw v5.1, JRefactory v2.6.34, JUnit v3.7, MapperXML v1.9.7, QuickUML 2001 and 33 industrial components
DPRE [42]	JHotDraw v5.1, Apache Ant v1.6.2, JHotDraw v6.0b1, QuickUML 2001, Swing and Eclipse JDT components (Core v3.3.3 and User Interface v3.3.2)
MARPLE [43]	30 open source projects
Sempatrec [44]	JHotDraw v5.1, JRefactory v2.6.24 and JUnit v3.7
KT [45]	KT and three Smalltalk programs
DP++ [46]	DTK library
Kim and Boldyreff [47]	Three systems (no information about them)
Heuzeroth *et al.* [48]	Java swing
Philippow *et al.* [49]	Students' projects
HEDGEHOG [50]	AJP code example, pattern box and Java language (v1.1 and v1.2)
Kaczor *et al.* [51]	JHotDraw v5.1, QuickUML 2001 and Juzzle

Table 2.5: Summary of the subject systems used by detection approaches

2.3.3 Recovered Design Patterns

Figure 2.13 shows a summary of the recovered design patterns detected by different detection approaches. Most approaches successfully detect the Composite design pattern because its structure is easy to detect. On the other hand, the Memento and Interpreter design patterns are only detected by three approaches, since they require dynamic analysis capabilities to detect them. However, most detection approaches focused on a specific set of design patterns.

Moreover, as Figure 2.13 illustrates, only three approaches successfully detect all GoF design patterns. Specifically, Kim and Boldyreff [47] recovered all GoF design patterns from three systems, programmed using C++. Unfortunately, there is no information on these systems. In addition, Philippow *et al.* [49] recovered all GoF design patterns from student projects, also programmed using C++. The main disadvantage of the previous two approaches is their results validation in the sense that the authors did not report how the detected design instances were validated. The third approach that recovers all GoF design patterns is presented by Dongjin *et al.* [33]. This approach recovers design patterns from Java open source projects using sub-patterns and method signatures. The Dongjin *et al.* approach used the repository of Perceron [34] as a reference benchmark to validate the detected instances. However, contradictions in the experimental results were noticed.

R DP	[18]	[19]	[20]	[22]	[23]	[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33][47][49]	[35]	[36][37]	[38]	[39]	[40]	[41]	[42]	[43]	[44]	[45]	[46]	[48]	[50]	[51]
SI	✓	✓	✓						✓				✓		✓		✓			✓	✓		✓	✓				✓	
FM	✓	✓	✓	✓								✓	✓		✓		✓			✓	✓		✓	✓				✓	
AF		✓			✓						✓				✓					✓			✓	✓				✓	✓
BU		✓										✓			✓								✓						
PR												✓	✓		✓		✓				✓		✓					✓	
AD	✓						✓	✓	✓				✓	✓	✓	✓	✓			✓	✓	✓	✓	✓		✓		✓	
BR				✓		✓	✓	✓		✓		✓		✓	✓	✓				✓		✓						✓	
CO	✓					✓		✓		✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓
DE			✓					✓					✓		✓	✓	✓		✓	✓	✓	✓		✓	✓		✓	✓	
FA								✓							✓					✓		✓							
FL								✓							✓					✓						✓		✓	
PR	✓							✓				✓			✓	✓				✓		✓	✓					✓	
CoR															✓					✓							✓		
CM													✓		✓		✓				✓								
IT											✓				✓													✓	
IN															✓														
ME															✓			✓		✓							✓		
MN															✓														
OB	✓		✓										✓		✓		✓			✓	✓			✓			✓	✓	
ST									✓				✓		✓		✓			✓	✓			✓				✓	
SR						✓			✓	✓		✓	✓	✓	✓		✓			✓	✓			✓				✓	
TM			✓	✓					✓			✓	✓		✓		✓			✓	✓			✓	✓			✓	
VI	✓												✓		✓		✓			✓	✓			✓			✓	✓	
Total	7	4	5	3	1	3	2	7	5	3	3	7	12	4	23	5	12	2	1	17	12	6	8	11	3	3	5	16	2

Note:

PR: Prototype	FM: Factory Method	BU: Builder	AD: Adapter	SI: Singleton
BR: Bridge	DE: Decorator	PX: Proxy	CO: Composite	FA: Façade
FL: Flyweight	VI: Visitor	TM: Template Method		CM: Command
OB: Observer	ST: State	SR: Strategy	IT: Iterator	AF: Abstract Factory
CoR: Chain of Responsibility		ME: Mediator	MN: Memento	IN: Interpreter
R: Reference	DP: Design Pattern			

Figure 2.13: Summary of design patterns recovered by detection approaches

2.3.4 Evaluation Criteria

Precision and recall metrics have been used by most detection approaches to evaluate their accuracy. A few approaches reported the F-measure which provides the harmonic means of recall and precision. Accuracy varies from one approach to another since some approaches recovered a few patterns and achieved high precision. The validation method, pattern definitions and pattern variants could also affect the detection accuracy. The precision, recall and F-measure have been calculated as follows [52]:

Precision = [True Positives / (True Positives + False Positives)] %

Recall = [True Positives / (True Positives + False Negatives)] %

F-measure = 2 × [(Precision × Recall) / (Precision + Recall)] %

Where:

True positives: the number of instances which are correctly detected.

False positives: the number of instances which are incorrectly detected.

False negatives: the number of instances which are incorrectly rejected.

The reported accuracy for the majority of detection approaches in the literature is presented in Table 2.6.

As Table 2.6 illustrates, the reported accuracy for most of the detection approaches is not balanced (i.e. high precision and low recall or vice-versa). The main reason for this would be the large differences between the number of correctly detected instances and the number of rejected instances. Specifically, the unbalanced accuracy suggests that there are no trade-offs between the number of correctly detected instances and the number of rejected instances (missed instances).

Some approaches only reported the number of true positives and true negatives, such as D3 [19], Marek Vokac [20] and Heuzeroth *et al.* [48]. On the other hand, some approaches used CPU times, such as Rasool *et al.* [18], DP-Miner [32], CrocoPat [38], SPQR [39] and PINOT [40], to evaluate their detection efficiency. For example, PINOT spent 66.79 seconds, 8.98 seconds, 10.68 seconds and 12.58 seconds detecting design pattern instances from Swing, JHotDraw, Java AWT and Ant respectively.

Furthermore, some detection approaches validated their results based on manual tracing of the source code and these achieved high accuracy. On the other hand, only two approaches, Dongjin *et al.* [33] and Sempatrec [44], validated their results based on design pattern repositories, such as the repository of Perceron [34]. Consequently, different accuracy values were achieved by different approaches since there is no standard benchmark to validate the recovered design pattern instances.

Tool/ Author	**Precision %**	**Recall %**
Rasool *et al.* [18]	94	92
Antoniol *et al.* [25]	30	Not Mentioned
Uchiyama *et al.* [27]	63	76
SSA [31]	100	66.7-100
Dongjin *et al.* [33]	68-100	73-100
Pat [35]	14-50	Not Mentioned
DeMIMA [41]	34	100
DPRE [42]	62-67	Not Mentioned
MARPLE [43]	76	63
Sempatrec [44]	61-82	88-90
Kim and Boldyreff [47]	43	Not Mentioned
HEDGEHOG [50]	100	85
D3 [19], Marek Vokac [20], SPOOL [22], MAISA [23], FUJAPA [24], Detten and Becker [26], Seemann and Gudenberg [28], DEPAIC++ [29], Columbus [30], DP-Miner [32], PTIDEJ [36][37], CrocoPat [38], SPQR [39], PINOT [40], KT [45], DP++ [46], Heuzeroth *et al.* [48], Philippow *et al.* [49], Kaczor *et al.* [51]	Not Mentioned	Not Mentioned

Table 2.6: Summary of reported accuracy by detection approaches

2.4 Reported Impact of Design Patterns

Many factors influence the quality of software systems. One of these factors is the implementation of design patterns. However, since their introduction in 1995, design patterns' impact on software quality is not well investigated. Some studies claim that the implementation of certain design patterns has a positive impact on the quality of software systems. In contrast, other studies claim that the implementation of the same design patterns has a negative impact on software quality.

One of the first studies to investigate the impacts of design patterns on software quality was conducted by Lange and Nakamura [53]. They concluded that design patterns enhance program understandability since they can serve as a guide in program exploration and thus make the process of understanding more efficient. Lange and Nakamura showed that design patterns can help in two different ways: recognized at a certain point in the process of understandability, they can fill in the blanks and they can act as starting points for exploring a given system. Lange and Nakamura's study was limited to a single quality attribute and to a few patterns. The study concluded that the informal and semantically rich nature of design patterns prohibits the creation of the exact description required for automation.

The study presented by Wendorff [54] reports on a large commercial project where the uncontrolled use of design patterns has contributed to maintenance problems. By using qualitative arguments, Wendorff's study showed that design patterns affect different quality attributes and that the application of design patterns may, for example, result in a desirable increase of flexibility at the cost of an undesirable increase in complexity. Wendorff suggests that a design pattern will usually add to a particular aspect of flexibility, but cannot provide universal flexibility. If a pattern is applied to enhance the flexibility of a software design, careful judgment is required to ensure that the pattern promotes the desired aspect of flexibility. Hence, it can make economic sense to remove an inappropriate design pattern from a source code. Wendorff found two categories of inappropriately applied design patterns in the subject system. The first category involves patterns misused by software developers who did not understand the rationale behind the patterns. The second category is the patterns that do not fall into the first category, but which do not match the system's requirements. Wendorff presented a procedure of seven steps to identify, assess and remove design patterns. This procedure has led to more objective, well-documented and economically decisions during reengineering activities.

Wydaeghe *et al.* [55] presented a study of the development of OMT editor using Observer, Visitor, Module-View-Controller, Iterator, Facade, Bridge and Chain of Responsibility patterns. They discussed the impact of these patterns on modularity, reusability, flexibility and understandability. The impact was reported as either positive or negative or as having no impact. Wydaeghe and the research team concluded that not all design patterns had a positive impact on quality attributes. For example, the Wydaeghe study claims that structural design patterns make the OMT editor more modular, while behavioral design patterns do not affect this property. Concerning understandability, structural design patterns increase the understandability of the OMT editor, while behavioral design patterns make the application less understandable. Furthermore, all the implemented design patterns increase the flexibility of the OMT editor. However, the Wydaeghe *et al.* study can hardly be generalized to another context of development.

McNatt and Bieman examine the notation of patterns coupling to classify how design patterns may include coupled patterns [56]. They classified the set of connected patterns in terms of loose and tight coupling. In addition, McNatt and Bieman classified three types of patterns interaction: intersection, composite and embedded. They showed that, when patterns are loosely coupled and abstracted, maintainability, factorability and reusability are well supported by the patterns. The study concluded that there was a need for more studies to investigate the effects of software design patterns on quality.

The study presented by Ellis *et al.* [57] shows that creating objects from factories used in Application Programming Interfaces (APIs) is significantly more time-consuming than creating objects from constructors, regardless of the context or the level of experience of the programmer using the API. The key goal of the Ellis *et al.* study is to provide quantitative measurements of the differences in usability between factories and constructors. The results are collected after asking a number of participants to perform certain tasks. The study concluded that the Factory pattern erodes the usability of APIs in which it is used.

The experiment presented by Hannemann and Kiczales develops and compares Java and AspectJ implementations for the 23 GoF design patterns [58]. AspectJ is a seamless, aspect-oriented extension to Java, which means that programming in AspectJ is effectively programming in Java plus aspects. The results show that using AspectJ improves the implementation of many GoF design patterns. For each of the 23 GoF design patterns, Hannemann and Kiczales created a small example that makes use of the pattern and then implemented the example in both Java and AspectJ. For a number of patterns, the AspectJ implementations show several closely related modularity benefits, such as locality, reusability, dependency inversion, transparent composability and unplugability. The AspectJ implementations of 17 of the 23 GoF design patterns were localized. The improvement in the AspectJ implementations are due to the inverting dependencies, so that the pattern code depends on the participants. Specifically, in AspectJ implementation, all codes related to a particular pattern instance are contained in a single module. Reusable pattern implementations have been developed by generalizing the roles, pattern code, communication protocols and relevant conceptual operations in abstract reusable aspects. The experiment concluded that the improvement using AspectJ in pattern implementations is directly correlated to the presence of cross-cutting structures in the patterns. This cross-cutting structure arises in patterns that superimpose behavior on their participants (superimposed roles are often interfaces that define behavior and responsibilities).

The study presented by Jeanmart and Guéhéneuc aims to determine whether the use of the Visitor design pattern is useful for maintenance through comprehension and modification tasks [59]. The study compared the developer’s efforts in the presence or not of the Visitor design pattern when performing comprehension and modification tasks and when using different layouts of the Visitor design pattern. The experiments have been conducted to collect data with which to compare the developers’ efforts when performing comprehension and modification tasks using different semantically equivalent UML class diagrams. Effort function was defined as the amount of attention that developers must spend to perform the tasks: less attention and less time means less effort. Jeanmart and Guéhéneuc captured the developers' attention using the data collected with an eye-tracker to decide whether a class diagram decreased their efforts with respect to the others. Data were collected for JHotDraw, JRefactory, PADL and 24 developers. For comprehension tasks, the results show that no significant difference exists between class diagrams with or without the Visitor design pattern and with a modified representation of the Visitor design pattern. For modification tasks, the results found that developers performed with significantly less effort on diagrams where the Visitor is represented in its standard structural format presented by GoF [11].

In [60], the use of two design patterns, Visitor and Decorator, to automate the validation of class invariants in C++ applications is described. An invariant on class C is a set of Boolean conditions that every instance of C will satisfy after instantiation and before and after every method invocation by another object. Class variants, expressed in Object Constraint Language (OCL), were used to ensure that the operations performed on instances of the class maintained the integrity constraints of that class. These constraints were described in terms of the member functions and data attributes of the class. A case study is presented of invariant validation in Keystone which is a parser and front-end for C++. Quantitative results are presented to measure the impact of these approaches on the case study. The results show that the use of Visitor and Decorator design patterns provide flexibility in terms of the frequency and level of granularity of validation of the class invariants.

Ampatzoglou and Chatzigeorgiou [61] performed a qualitative and quantitative evaluation of two open source projects to evaluate the use of design patterns in game development. For the quantitative evaluation, the projects are being analyzed by reverse engineering techniques and software metrics such as Lines of Code (LOC), Number Of Classes (NOC), Attribute Complexity (AC), Weighted Methods per Class (WMPC), Coupling Factor (CF) and Lack of Cohesion Of Methods (LCOM). The results indicate that design patterns can be beneficial with respect to maintainability. The game version that includes the subject pattern has reduced complexity and coupling compared with a version without the pattern. Furthermore, the implementation of design patterns tends to increase the cohesion of the software. In contrast, the size of the subject system has increased in the pattern version.

Aversano *et al.* [62] report and discuss results from an empirical study aimed at analyzing how design patterns change during a software system's lifetime and to what extent such changes cause modifications to other classes that are not part of the design pattern. The study has been performed on three Java open source systems, JHotDraw, ArgoUML and Eclipse-JDT. Design pattern instances have been detected from the three systems using SSA [31]. Then, changes from Concurrent Versioning System (CVS) have been mined to identify when a pattern changed, what kind of change was performed, which classes co-changed with the pattern, whether these classes had a dependency to or from the pattern and what the relationship was between the type of change made and the resulting co-change. Results indicate that the pattern change frequency and the amount of co-change do not depend on the pattern type, but rather on the role played by the pattern to support the application features.

Bieman *et al.* [63] studied five systems, three proprietary systems and two open source systems, to identify the observable effects of the use of design patterns on changes that occurred as the system evolved. In particular, the study was aimed at determining whether software with design patterns tended to be adapted by creating new concrete classes that were extensions of existing pattern classes, interfaces, or abstract classes, or by modifying existing pattern classes. Furthermore, the study looked at the relationship between design structure and software changes. The design structure was characterized by class-size and class participation in inheritance relationships and design patterns. Changes were measured in terms of a count of the number of times that a class was modified over a period of time. Bieman *et al.* quantified the design structure of an early version of each system and studied the relationship between design attributes of this version and future system changes. The results showed that classes playing certain roles in some design patterns were more change prone than other classes in the subject system. An informal analysis suggests that pattern- participant classes provide a key functionality to the system, which may explain why these classes tend to be modified relatively often.

An empirical study presented by Di Penta *et al.* [64] aims to understand whether there are design pattern roles that are more change-prone than others and whether there are changes that are more likely to occur to certain roles. The investigated changes are changes to method implementation, method addition/removal, attribute addition/removal and extension by sub-classing. The results on three open source systems, JHotDraw, Xerces-J and Eclipse-JDT, show that classes playing certain roles in design patterns are more change prone than are other classes. For example, in the Adapter design pattern, classes playing the role of Adapter are more change prone than other Adapter participant classes. In addition, the obtained results suggest to carefully design roles that are more subjects to changes, since their change proneness can make other parts of the system less robust to changes.

In a replication of the Bieman *et al.* study [63], Gatrell *et al.* [65] examined a commercial C# consisting of 7439 classes. This system had been subject to 19054 changes over a two-year period and these changes were caused by both enhancements and fault fixing. The pattern participation characteristics were compared with the change history of the classes to determine any relationships. Each modification in the version-control system, whether for a fault-fix or an enhancement, resulted in a new version of the class and each version was counted as a single change. Results were found to support the earlier study: classes participating in design patterns were found to have changed more frequently than other classes in the system. In addition, the study went further to show that the Adapter, Template method, Proxy, Singleton, State, Strategy and Visitor patterns caused the highest rate of change. On the other hand, classes participating in the Command and Creator design patterns had a relatively lower rate of change.

The study presented by Baudry *et al.* [66] explains how the use of design patterns can provide a way to limit the complexity of testing for conflicts, and to limit design pattern effects to the classes involved in the pattern. The study focused on testing problems that appeared at system level as a result of interactions among classes and also of polymorphism. The study suggests that the pattern-refined design is more testable than the "classical" design since the complex hierarchical control structure of the classical design has been removed.

The contribution of the Baudry *et al.* study in [67] concerns both a given metric and the practical way to apply it in the usual object-oriented design process. The study aims to show how to integrate testability improvements into the usual design process. In addition, the study addresses two configurations of object-oriented anti-patterns that can weaken its testability. It shows how testing risks might be avoided using two risk mitigation techniques: a guideline on the risk for applying a pattern, called the testability grid, and also design refinement constraining.

Elish in [68] discusses with examples the impact of four structural design patterns (Adapter, Bridge, Composite and Façade) on the stability of class diagrams; its resistance to the propagation of changes. The examples used were adapted from examples provided by Design Patterns in Java: Reference and Example Site [69]. Modifications made to one class can have ripple effects on other classes in the diagram. A good class diagram, from standpoint of stability, should localize changes as much as possible, confining them to those classes where changes are made. The examples presented show that the Adapter design pattern has a positive impact on the stability of class diagrams as it enables the client and the adaptee participant classes to be completely decoupled from each other. Changes will be localized to the adapter class and will not propagate to other classes in the diagram. Furthermore, the examples presented show that the Bridge design pattern enhances the stability of class diagrams, since the abstraction and implementor hierarchies can be extended independently, and also shows that modifying an implementation class does not require the recompiling of the abstraction class. Concerning the Composite design pattern, the client participant class uses composite and leaf classes uniformly, which permits the adding of new kinds of composite and leaf classes to the hierarchy without affecting the client participant class. The client does not need to be changed for new composite and leaf classes. Finally, the Façade design pattern supports the stability of software since changes made to one of the sub-system classes cannot be propagated beyond the Façade class. Hence, changes are localized within the sub-system. The study concluded that Adapter, Bridge, Composite and Façade have a positive impact on the stability of class diagrams. However, an empirical evaluation is still needed to confirm the achieved results.

Khomh and Guéhéneuc [70] studied the impact of design patterns on quality attributes in the context of software maintenance and evolution. An empirical study using a questionnaire was conducted. The quality attributes addressed were Expandability, Simplicity, Reusability, Learnability, Understandability, Modularity, Generality, Modularity at runtime, Scalability and Robustness. Each quality attribute was evaluated using a six-point Likert scale: A - Very Positive, B - Positive, C - Not Significant, D - Negative, E - Very Negative and F - Not Applicable. The questionnaires of 20 software engineers were selected for evaluation since they had verifiable experience in the use of design patterns in software development and maintenance. The respondents considered that, although design patterns were useful for solving design problems, they did not always improve the quality of systems in which they were applied. Most respondents considered that design patterns decreased simplicity, learnability and understandability. In addition, the study showed that design patterns negatively affected several quality attributes. It concluded that design patterns should be used with caution during the development process since they impeded maintenance and evolution.

The experiments of Prechelt *et al.* [71] investigated software maintenance scenarios that employed various design patterns and compared them with other, simpler designs. Professional software engineers were used as subjects. The design patterns addressed were Visitor, Observer, Abstract Factory and Decorator. The maintenance tasks were applied on Stock Ticker, Boolean Formulas, Communication Channels and Graphics Library. In most of the nine maintenance tasks tested, design patterns had a positive impact.

Research by Vokac *et al.* [72] investigates when, and how, using design patterns is beneficial and whether some design patterns are more difficult to use than others. The experiments were conducted to test whether design pattern P does, or does not, improve the performance of subjects doing maintenance-work task X on program A (containing P) when compared with subjects doing the same task X on an alternative program A' (not containing P). The design patterns addressed are Visitor, Decorator, Observer and Abstract Factory. The study concluded that design patterns were not universally good or bad, but must be used in a way that matched the problem. The Observer and Decorator design patterns were understood by subjects with little or no previous pattern knowledge.

Kouskouras *et al.* [73] investigated the behavior of an object-oriented software application at a specific extension scenario, following three implementation alternatives with regard to a certain design problem relevant to the extension. These implementations were a simplistic implementation, a design pattern implementation and Aspect-Oriented implementation. The study identified the additional design implementations needed to perform the extension and evaluated the effect of the extension on several quality attributes. Each implementation was assessed by exploring qualitative aspects, supported by observations of the design implications, and also quantitative aspects, supported by specific metrics values before and after extension. The metrics were calculated at both class level and package level. An emulator was developed to allow the user to configure it with commands and to perform simple traffic cases. The design pattern implementation showed that the coupling metrics on the class level for the relevant classes were increased. In contrast, these couplings did not have any significant effect on the inter-package dependencies

In [74], a concurrent design pattern framework has been presented to unify program design for modularity with program design for concurrency.

The concurrent design pattern framework provided enhanced versions of GoF patterns for Java programs. The study addressed all 23 GoF design patterns and found that, for 18 patterns, synergy between modularity goals and concurrency goals was achievable. The presented framework relied on Java's existing type system and libraries to enforce concurrency and synchronization discipline. The study concluded with an understanding that a sophisticated runtime system as a back-end would be necessary to abstract completely from the concurrency concern. In addition, performance evaluation of several design patterns suggested the need to support load-balancing in the developed framework.

Most previous studies in the literature used experiment [57], [58], [59] and [60], case studies [56], [60], [61], [62] and [63], conceptual analysis [64], [65] and [66] and survey [70] to assess the impact of design patterns on software quality attributes. We believe that these methods lead to controversial results since most are based on human intervention and lack accuracy. Table 2.7 summarizes the reported impact, to the best of our knowledge, of GoF design patterns on software understandability and maintainability.

As Table 2.7 demonstrates, 18 out of 23 design patterns have been reported to have a positive impact on software maintainability. More specifically, creational and structural design patterns positively affect software maintainability, while the impact of behavioral patterns is controversial. Concerning understandability, the results are controversial for all pattern categories.

	Patterns/Quality Attributes	Understandability	Maintainability
Creational	Singleton	+[53],+[56], +[70]	-[70],-[72]
	Prototype	+[53], +[56], +[70]	+[70],+[74]
	AbstractFactory	+[53], +[56], -[57], -[70],-[72]	+[70],+[74]
	Factory method	+[53], +[56], -[70]	+[70],+[74]
	Builder	+[53], +[56], +[70]	+[70],+[74]
Structural	Adapter	+[53], +[56], -[70]	+[70],+[74]
	Bridge	+[53], +[56], +[70]	+[61],+[70],-[74]
	Composite	+[53], +[56], +[70],+[72]	+[70],+[74]
	Decorator	+[53], +[56], -[55], -[70],+[71], +[72]	+[70],+[54],+[71],+[55],+[72]
	Façade	+[53], +[56], +[70]	+[70],+[74]
	Flyweight	+[53], +[56], -[70]	-[70],-[74]
	Proxy	+[53], +[54], +[56],-[70]	-[54], -[70],+[73],+[74]
Behavioural	CoR	+[53], +[56], +[70]	+[70]
	Command	+[53], +[56], -[70]	+[70],+[74]
	Interpreter	+[53], +[56], +[70]	+[70], +[74]
	Iterator	+[53], +[56], +[70]	+[70],+[74]
	Mediator	+[53], +[56], +[70]	+[70],+[74]
	Memento	+[53], +[56], -[70]	-[70],-[74]
	Observer	+[53], +[56], -[70],-[71],+[72]	+[70], +[74]
	State/Strategy	+[53], +[56], +[70]	+[61], +[70],-[74]
	Visitor	,[48],-[51],+[53], +[56], +[70], -[72]	+[59], +[70],+[54], +[71],+[74], -[48],-[51],-[72]
	Template Method	+[53], +[56], -[70]	+[70],+[74]
	+[RX]: Reference X claims that the pattern positively affects the corresponding quality **-[RX]**: Reference X claims that the pattern negatively affects the corresponding quality		

Table 2.7: Reported impact of design patterns on understandability and maintainability

2.5 Lessons Learned

This chapter presented a comprehensive comparison between different design pattern detection approaches and the reported impact of design patterns. The lessons learned can be summarized as follows:

- Design patterns are described from different perspectives by different approaches, such as structural aspects, behavioral aspects, and semantic aspects.
- Current detection approaches use different tools to get the intermediate representation of the subject source code. This will directly affect the recovery process.

- The discovery tool of each approach only supports the discovery of specific patterns. Only a few approaches successfully detected all GoF design patterns.
- Different approaches conduct experiments on different open source systems.
- Recall and precision were used to evaluate the accuracy of the detection process. Only a few approaches reported F-measure, such as Dongjin *et al.* [33] and Sempatrec [44]. In addition, some approaches measured the CPU times and memory consumptions to evaluate their detection efficiency.
- There is no standard benchmark to validate the recovered design pattern instances. The available benchmarks, to the best of our knowledge, are the repository of Perceron [34], the Design Pattern Detection tools benchmark platform [75], P-MARt [76] and BEFRIEND [77].
- Design patterns' impact on software quality is not well investigated. Most previous studies in the literature used experiment, case studies, conceptual analysis and survey to assess the impact of design patterns on software quality attributes.

2.6 Summary

This chapter presented the current state of the art of design pattern detection approaches and the current reported impact of design patterns. Specifically, we presented a comparative study on design pattern detection approaches in terms of detection methodology, analysis style, system representation, subject systems, recovered design patterns and evaluation criteria. The key contribution of this chapter is the necessity to address all detection approaches and tools. This will guide future researchers in developing more accurate detection tools. In addition, this chapter will facilitate the comparison between different detection approaches and any new detection approach, since there is no trusted benchmark to evaluate the recovered design pattern instances. Most design pattern detection approaches target open source systems that do not have proper documentation. It could be worthwhile to conduct the experiments on industrial and commercial applications. In addition, disparity among the results is noticed. The main reason could be the missing roles and the implementation variants of design patterns. Precision and recall were used to evaluate the accuracy of the detection process. However, the reported accuracy is not balanced (i.e. high precision and low recall or vice versa). One possible solution is to use the common formalized definition of GoF patterns. All detection approaches are working independently without any ability to integrate them together. The research community should make efforts to build new approaches which may be integrated with other existing approaches.

Finally, some studies claim that the implementation of certain design patterns has a positive impact on the quality. In contrast, other studies claim that the implementation of the same design patterns has a negative impact. Hence, the current reported impact of design patterns is controversial.

Chapter Three
Methodological Approach

The detection of design patterns is a reverse engineering activity where design patterns are recovered depending on certain criteria. This chapter presents a Multiple Levels Detection Approach (MLDA) to recover the instances of GoF design patterns from the Java source code. MLDA aims to recover design pattern instances with reasonable detection accuracy. Moreover, MLDA introduces a rule-based approach to filter the candidate design instances by matching the method signatures of the candidate design instances to that of the subject system.

3.1 Introduction

Design patterns provide template solutions for certain design problems. The occurrence of design pattern instances in the source code reflects the earliest set of design decisions. Each design pattern has its own structural and behavioral aspects. The structural aspects concern the static arrangement of classes and interfaces. On the other hand, the behavioral aspects are concerned with the dynamic interactions between classes and interfaces.

Recovering design pattern instances from the source code requires representing it in one of the parsing formats, searching for all possible pattern structures and trying to fingerprint each structure with a certain behavior. The possible variants of a design pattern can complicate the detection process. These variants, non-standard implementations of design patterns, are difficult to capture since there is no reference benchmark to decide whether the recovered variant is a correct instance or not.

Design pattern information helps the system analyst, software engineer and software architect to capture design and code information and enhance the program understanding. In addition, design pattern information improves software documentations, captures expert knowledge and design trade-offs and helps in re-structuring the systems.

Current design pattern detection approaches only recover a specific pattern, or a few sets of patterns. Only three approaches, [33], [47] and [49], recovered all GoF design patterns. Furthermore, only one approach, the sub-patterns approach [33], uses the method signatures of the candidate design instances to detect design patterns. The sub-pattern approach did not explain how the matching of the method signatures of the candidate instance to that of the subject system is performed.

This chapter presents a Multiple Levels Detection Approach (MLDA) to recover the instances of design patterns from the Java source code. MLDA involves three levels: a parsing level, a searching level and a method signatures matching level. Each level performs certain tasks to collect the required information to recover GoF design pattern instances. This information consists of:

- Pattern participant classes
- Relationships between participant classes
- Methods inside each participant class
- Method calls between participant classes

More specifically, MLDA uses static analysis capabilities to recover instances of GoF design patterns using structural and method signature features. MLDA's static analysis capabilities can record objects' creation by recovering association and aggregation relationships. Dynamic and semantic analysis both require more sophisticated implementation of certain packages and record message interactions between classes during runtime. MLDA will recover instances of design patterns by building the structure of each design pattern and then matching the method signatures of the candidate instances to that of the subject system.

3.2 Research Methodology

The research methodology adopted in this thesis consists of three main steps. We followed these step to come up with our proposed methodology. These steps are representing the subject system and GoF design patterns, matching criteria and filtering criteria. Each step is described below.

The first step to recover the instances of design patterns is to represent the subject system and GoF design patterns in one of the parsing formats (such as abstract syntax graph or abstract syntax tree). These formats should store the structural aspects of the subject systems and GoF design patterns. Different structural aspects can be recovered such as the connecting relationships between classes and interfaces, abstract classes, concrete classes, attributes and methods. Hence, the recovering methodology should recover all or some of these structural aspects to recover the structures that are similar to that of GoF design patterns.

As a result, we modified a Javaparser in such way it can recover the five key relationships (Realization, Inheritance, Aggregation, Association and Dependency) that may occur between classes and interfaces inside any object-oriented program.

The second step to recover the instances of design patterns is to adopt a matching criteria between the representation of GoF design patterns and the representation of the subject system. The matching criteria should search for all the possible structures that are similar to that of GoF (i.e. searching for all possible arrangements of classes and interfaces that are similar to that of GoF). Consequently, the proposed methodology in this thesis introduces a Structural Search Model (SSM) which is able to recover the instances of design patterns based on the generated class level representation of the Java source code. Moreover, the SSM builds the structure of each design pattern incrementally. The SSM recovers the instances of design patterns based on the five key relationships that may occur between classes and interfaces inside any object-oriented program: Aggregation, Association, Dependency, Inheritance and Realization.

Finally, the recovering methodology should match the behavioral aspects of GoF design patterns to that of the subject system. These behavioral aspects consist of the required interactions between pattern participant classes (such as messages between participant classes and method signatures). This thesis focuses on the method signatures of GoF design patterns to reflect the required behavioral aspects of GoF Design patterns. The proposed methodology in this thesis introduces a rule-based approach to filter the candidate design pattern instances detected by the SSM. The rule-based approach matches the method signatures of the candidate design instances to that of the subject system.

3.3 The Catalog of Design Patterns

The catalog of design patterns presented by GoF involves 23 design patterns. Each design pattern has its own intent, structure and participant classes (roles) and provides a solution to a specific design problem. Pattern participant classes may implement one or more methods and may call on other methods implemented in other participant classes.

3.3.1 Design Pattern Elements

GoF suggested four essential elements for a design pattern:

- Pattern name, selected in such way that it provides a real indication of the intent and the goal of the pattern.
- The problem, describing a specific design problem and when to apply the pattern. This part may also suggest certain conditions that must be met before applying the pattern.
- The solution, describing the participant classes that build up a design pattern.

- The consequences, presenting the results and the trade-offs that come from applying the pattern. The consequences of applying a design pattern often concern time trade-offs and implementation issues and pattern's effect on the system's quality attributes.

3.3.2 Design Patterns Classification

Based on the intent behind the use of design patterns, GoF have classified them into three groups: creational patterns which concern the initialization of classes and objects; structural patterns which concern the composition of classes and objects; and behavioral patterns which concern the dynamic interaction between classes and objects. Another classification of design patterns is also presented by GoF, based on the scope of the pattern where design patterns have been classified into class patterns and object patterns.

Furthermore, each design pattern has been represented using a UML class diagram. Table 3.1 presents the catalog of GoF design patterns and their intents [11]. UML class diagrams for all GoF design patterns are presented in Appendix A.

The catalog of GoF design patterns presents a unique intent for each pattern. The intent can be achieved by implementing the required static arrangement of pattern participant classes. In addition, the behavior of the pattern participant classes should be implemented. Most GoF design patterns require method interactions between their participant classes. Each participant class must implement specific method(s) with certain signatures. An instance of a design pattern is said to be a complete instance if it implements all the required participant classes in addition to all methods. This thesis focuses on the detection of a complete structure of design patterns presented by GoF (standard implementation of design patterns).

The level of abstractions varies between design patterns. Some design patterns require three levels of class hierarchies, such as the Composite design pattern, to achieve its intent. In contrast, other design patterns require two levels of class hierarchies, such as the Template method, to achieve their intent. However, GoF's catalog does not suggest that design patterns are finished designs. More specifically, design patterns should be implemented as an integral part of the whole system design.

Type	Pattern Name	Intent
Creational Patterns	Singleton	Ensuring that each class has one instance
	Factory Method	Method in a derived class creates associates
	Abstract Factory	Factory for building related objects
	Builder	Building complex objects incrementally
	Prototype	Cloning new instances from a prototype
Structural Patterns	Adapter	Translator that adapts a server interface for a client
	Bridge	Abstraction for binding one of many implementations
	Composite	Structure for building recursive aggregations
	Decorator	Extending an object transparently
	Façade	Simplifies the interface for a subsystem
	Flyweight	Many fine-grained objects shared efficiently
	Proxy	One object approximates another
Behavioral Patterns	Chain of Responsibility	Request delegated to the responsible service provider
	Command	Request or Action is a first-class object, hence re-storable
	Iterator	Aggregate and access elements sequentially
	Interpreter	Language interpreter for a small grammar
	Mediator	Coordinating interactions between its associates
	Memento	Snapshot that captures and restores object states privately
	Observer	Dependents update automatically when the subject changes
	State	An object whose behavior depends on its state
	Strategy	Abstraction for selecting one of many algorithms
	Template Method	Algorithm with some steps supplied by a derived class
	Visitor	Operations applied to elements of a heterogeneous object structure

Table 3.1: The catalog of GoF design patterns

3.4 MLDA Architecture

Java programming language is the fundamental backbone of MLDA. The selection of Java was made because:

- Java provides many libraries and packages that are easy to import and modify. Importing Java packages and libraries provides quick solutions to common programming problems
- Java is a platform-independent language
- Java is a robust, secure, portable and high-performance language

Moreover, most of the detection approaches in the literature have been implemented in Java. Hence, the selection of Java to implement MLDA will facilitate the comparison with other existing approaches. However, implementing MLDA using other object-oriented programming languages may show a few differences in terms of accuracy and efficiency, since most object-oriented programming languages share similar properties. For example, the inheritance, association and aggregation relationships have the same intent in Java, C sharp and C++. Recovering design patterns requires a subject system representation, a design patterns library and matching criteria between the GoF catalog and the subject system's representation. MLDA uses the standard structural codes of design patterns presented by GoF to build a library representation for each design pattern. In addition, MLDA targeted the source code of the subject system since the design model does not provide any runtime data and most of the design models are not publically available. MLDA involves three levels: a parsing level, a searching level and a method signatures matching level. These levels work in a consistent and dependent manner. Figure 3.1 shows the architecture of MLDA.

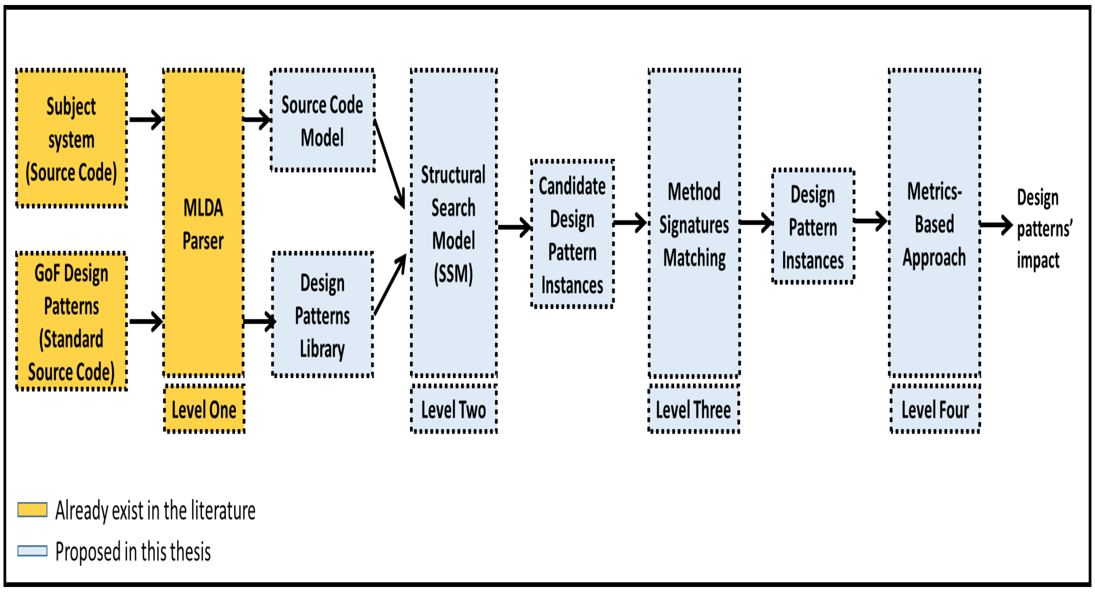

Subject system (Source Code)
GoF Design Patterns (Standard Source Code)
MLDA Parser
Level One
Source Code Model
Design Patterns Library
Structural Search Model (SSM)
Level Two
Candidate Design Pattern Instances
Method Signatures Matching
Level Three
Design Pattern Instances
Metrics-Based Approach
Level Four
Design patterns' impact
Already exist in the literature
Proposed in this thesis

Figure 3.1: The architecture of the proposed MLDA

MLDA aims to recover all GoF design patterns with reasonable detection accuracy in terms of precision and recall. Moreover, MLDA uses a rule-based approach to filter the candidate design instances by matching their method signatures to that of the subject system. MLDA recovers design pattern information from the source code of the subject system. This information concerns relationships between classes and their method signatures.

The parsing level aims to recover source code information and to generate a source code model. Specifically, the MLDA parser aims to recover the five key relationships that may occur between classes and objects inside any object-oriented program. These relationships are inheritance, aggregation, association, dependency and realization. In fact, MLDA provides a clear distinction between the aggregation relationship and the association relationship. In the aggregation relationship, the creation of the objects will occur during the compile time, while in the association relationship the creation of the objects will occur during the runtime. According to the GoF's catalog, the inheritance relationship is the main building unit of the structural design patterns. On the other hand, the association and aggregation relationships are the main building unit of the behavioral and creational design patterns. The searching level of MLDA aims to examine the source code model that has been generated during the parsing level and tries to match it with the GoF's catalog. Specifically, MLDA introduces a Structural Search Model (SSM) which involves a searching algorithm for each design pattern. MLDA works on the principle of building the pattern structure incrementally based on the connecting relationships. The third level of MLDA is the method signatures matching level. The method signatures of the subject system are represented as a set of facts. On the other hand, the required method signatures of the candidate design instances are represented as a set of rules. CLIPS (C Language Integrated Production System) [78], an expert system tool, has been used to match the generated facts and rules.

3.4.1 Parsing Level

Parsing is "the process of analyzing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar" [79]. MLDA's parsing level relies on the packages of the Javaparser version 1.0.11 which has been developed by Júlio Vilmar Gesser and is available online [80]. The Javaparser is an open source project and can be used under the terms of the LGPL licence. In fact, the Javaparser involves a number of useful packages, such as Japa.parser, Japa.parser.ast, Japa.parser.ast.expr and Japa.parser.ast.visitor. The motivation of importing the packages of Javaparser is their ability to generate an Abstract Syntax Tree (AST) that can record the source code structure. AST is a tree that represents the syntactic behavior of the source code, where its elements are mapped into tree nodes. The parsing level of MLDA aims to recover all the possible relationships between classes in the Java source code. Table 3.2 presents the relationships syntax which MLDA relies on to parse the source code of the subject system. The syntax of the relationships has been written based on the syntax presented in [81-86].

Relationships between class C_s and class C_d	**Common Java Syntax**

R (C_s, C_d) = {Inheritance}	public class C_s {
	...
	} // end C_s
	…..
	public class C_d extends C_s {
	
	} //end C_d

R (C_s, C_d) = {Dependency}	public class C_s {
	...
	public void doSomething (C_d b) { }
	…
	
	} // end C_s

R (C_s, C_d) = {Aggregation}	public class C_s {
	…
	private C_d _b;
	public void setB(C_d b) { _b = b; }
	…
	}// class Cs

R (C_s, C_d) = {Association}	public class C_s {
	…
	private C_d _b = new C_d ();
	
	…
	}//end C_s

	public class C_s {
	…
	private C_d _b;
	public void doSomethingUniqueToCd()
	{
	if (null == _b) { _b = new C_d (); }
	return _b.doSomething();
	} // doSomethingUniqueToCd()
	…
	}// class C_s

	public class C_s {
	…
	private C_d _b;
	public C_s () {
	_b = new C_d ();
	} // default constructor
	} //end C_s

R (C_s, C_d) = {Realization}	public interface C_s {
	...
	} // interface C_s
	public class C_d implements C_s {
	
	} // end interface C_d

Table 3.2: The relationships and their common syntax

The output of the parsing level is a model of the source code and a library of design patterns. The source code of the subject system is modelled in the form: source class, destination class and relationship type. The same structure is also applied to represent the catalog of GoF. The source code model generated by MLDA's parsing level is presented in Figure 3.2. This model will be exported into an SQL table which will be examined by the SSM in order to recover the candidate instances of design patterns. The library stores a representation of each design pattern. This representation is similar in its structure to that of the source code model (i.e. the representation of each design pattern in the library involves three columns: source class, destination class and relationship type).

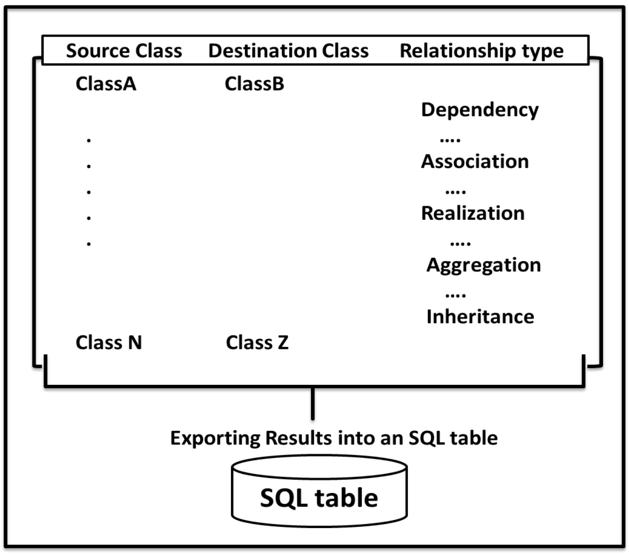


Figure 3.2: The source code model generated by MLDA

To explain how MLDA represents each design pattern in the library, the Command design pattern representation is presented in Figure 3.3. MLDA has successfully recovered two aggregation relationships and one inheritance relationship. One aggregation relationship is connecting the "Invoker" class to the "Command" class and the other is connecting the "ConcreteCommand" class to the "Receiver" class. Furthermore, the inheritance relationship between the "Command" class and the "ConcreteCommand" class is also recovered. However, MLDA has excluded the role of the "Client" class since it represents the role of the main program inside the source code. This will not affect how the Command participant classes are connected, or how they communicate together.

Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
ConcreteCommand	Receiver	AGGREGATION
ConcreteCommand	Command	INHERITANCE
Invoker	Command	AGGREGATION

Figure 3.3: The representation of the Command design pattern in the library

3.4.2 Searching Level

The searching level of MLDA aims to build the design pattern structure incrementally from the source code model based on its representation in the library. In addition, the searching level of MLDA introduces what is called the SSM which involves a searching algorithm for each GoF design pattern. Each part of the SSM involves two participant classes (i.e. source and destination classes). Specifically, the searching algorithm tries to build the pattern structure from the source code model by checking the relationship connecting the source class to the destination class. If the search process finds one of the required relationships of the pattern, it will continue searching for the remaining relationships until it can form a complete pattern structure similar to the pattern representation in the library. When the pattern structure has been found in the source code model, all pattern participant classes are exported from the source code model to an SQL table. MySQL Workbench version 6.3 CE was used to create the tables. Since MLDA is able to distinguish between the aggregation and association relationships and records all the object creations and the dynamic interactions between classes, it is expected to recover all behavioral design patterns.

To explain how the searching level works, the searching algorithms of the Builder, Proxy and Command design patterns are presented. Figure 3.4 shows the SSM of the Builder design pattern.

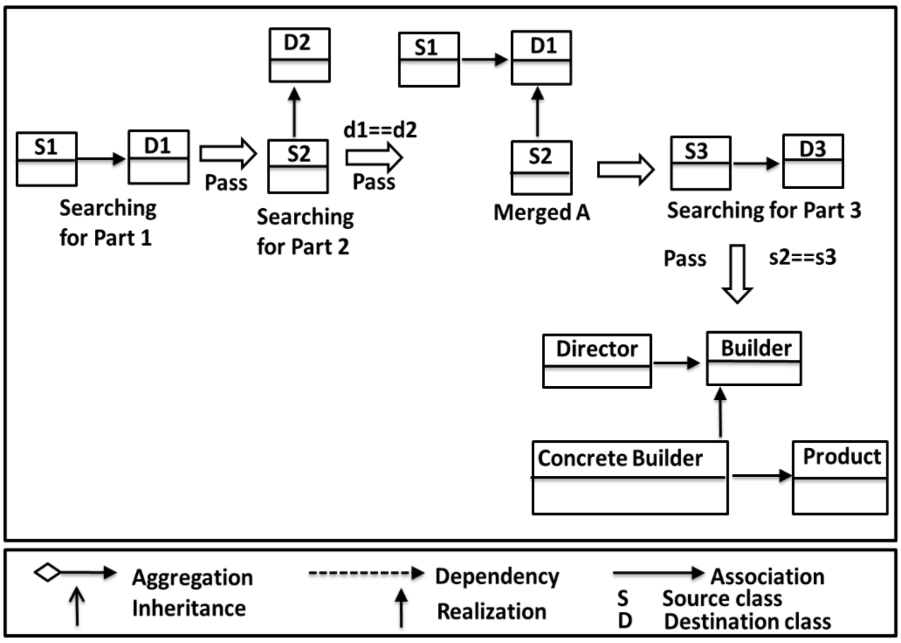


Figure 3.4: The structural search model of the Builder design pattern

MLDA will examine the source code model of the subject system and start the searching process by trying to find source and destination classes that are connected together by an association relationship (building the first part of the Builder design pattern). If the first part is successfully formed, MLDA will continue searching for the second part which needs a realization relationship between its participant classes. When a realization relationship is encountered, MLDA will try to combine the first and second parts together. The Builder class is acting as a connecting class (i.e. the first and second parts are combined together when the destination class of the first part is the same destination of the second part).

MLDA will continue searching for the remaining relationships. However, if MLDA is unable to combine the two parts together, the search process will terminate and MLDA moves to the next record in the table to start a new searching attempt. Finally, to form the Builder design pattern, MLDA will search for two participant classes that connect together via an association relationship and which differ from the participant classes of the first part. MLDA will combine the merged parts and the third part together if they have the same source classes. Each record of the source code model will be visited more than once as MLDA tries to build the parts incrementally.

Figure 3.5 shows the SSM of the Proxy design pattern. The recovery of Proxy instances is based on its representation in the library. MLDA will search for two participant classes that have a realization relationship. The retrieved classes are stored temporally for later use. Then MLDA will continue searching in the table, which represents the source code model, for another two classes that are also connected together by a realization relationship. MLDA will combine the two recovered parts together if they have the same superclass (root) and different subclasses.

If MLDA has successfully combined the two recovered parts, the process will continue searching for another two classes that are connected together using an association relationship. All classes form an instance of the Proxy design pattern if the role of the third part's source class is similar to that of the merged part's source class. In addition, the role of the destination class of the third part must be similar to the role of the source class of the first part. These conditions have been checked by SSM using nested IF-THEN statements. All the recovered instances and their participant classes will be exported to the SQL table.

Figure 3.6 presents the searching attempts to recover the instances of the Command design pattern. The Command design pattern involves four main roles: Invoker, Command, ConcreteCommand and Receiver. MLDA will start searching for two classes that are connected using an aggregation relationship (searching attempt one). The next searching attempt aims to recover another two classes connected together using an inheritance relationship (searching attempt two). MLDA will combine the parts of searching attempt one and searching attempt two together if they have the same destination class, consequently forming the "Merged A" part. Finally, MLDA will search for another two participant classes connected using an aggregation relationship and which differ from the classes recovered during the third searching attempt. If the second and third searching attempts have the same source class, then all classes together form an instance of the Command design pattern. The design pattern library generated by MLDA and the structural search model and its pseudocode for all GoF design patterns are presented in Appendices A, B and C respectively.

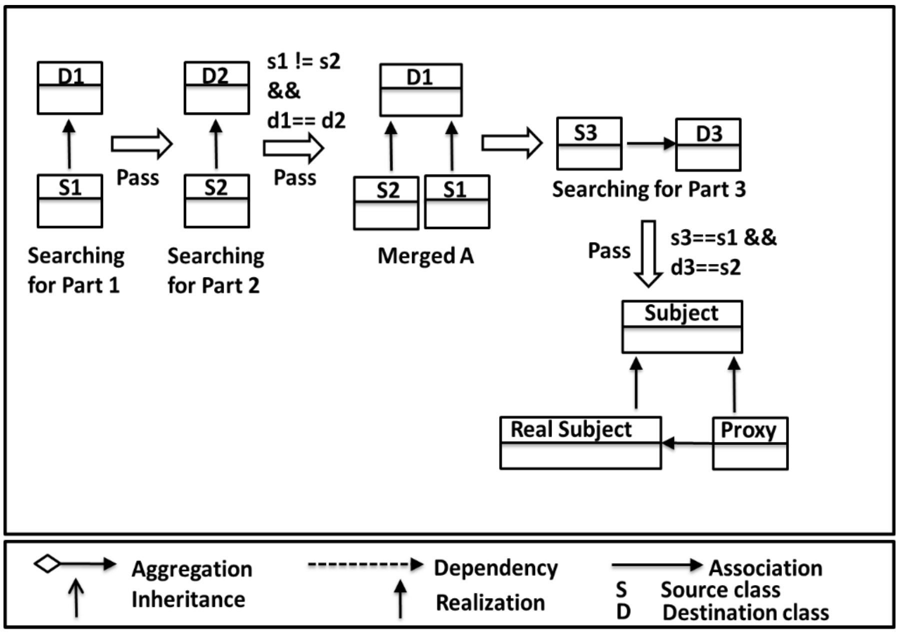


Figure 3.5: The structural search model of the Proxy design pattern

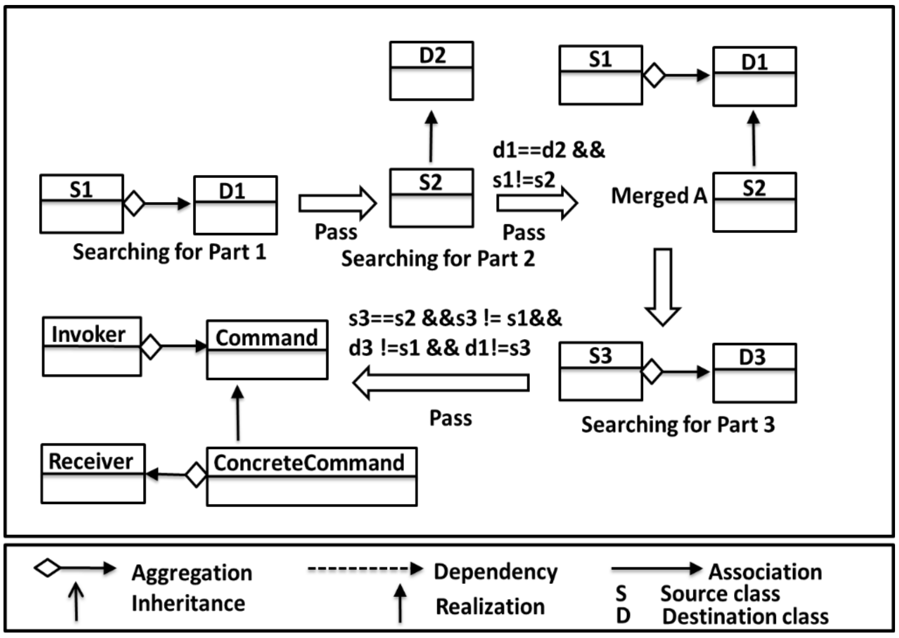


Figure 3.6: The structural search model of the Command design pattern

3.4.3 Method Signatures Matching Level

The candidate design pattern instances that have been detected by SSM will be filtered by applying a rule-based approach which aims to remove the false positive instances by matching the method signatures of the candidate design instances to that of the subject system. The MLDA parser will parse the subject system and recover all method signatures from each class/interface. The recovered signatures are access modifier, is_static, returntype and call_to. On the other hand, a rules template for GoF method signatures has been created to reflect the required method signatures for each design pattern. The created rules template relies on the standard structural definitions of design patterns presented by GoF [11].

In addition, the so-called MLDA rules/facts generator is developed, which is a simple Java program able to write a set of rules and facts based on the method signatures representation of the candidate design instances and subject system. Specifically, the MLDA rules/facts generator will generate a list of rules to reflect the required method signatures and method calls between candidate instance participant classes. On the other hand, the MLDA rules/facts generator will generate a list of facts to represent interactions between methods inside the subject system.

A rule based-system contains IF-THEN rules, facts and an inference engine that controls the application of the rules. Our main motivation behind the use of a rule-based approach is the ability to represent the method signatures of the candidate design instances as an independent piece of knowledge, which can be transformed into a set of rules. In addition, the method signatures of GoF design patterns have a uniform structure which facilitates their representation as a set of rules. In addition, the comparison process performed by the inference engine allows an effective match between the set of rules and the facts. MLDA uses CLIPS v6.3, an expert system tool, to process the generated facts and rules and to remove the false positive instances. Further details on MLDA's level three are presented in Chapter 5 and Chapter 6.

3.5 Summary

This chapter presented a Multiple Levels Detection Approach (MLDA) to recover design pattern instances from the Java source code.

MLDA works on three levels to recover the instances of GoF design patterns: a parsing level, a searching level and a method signatures matching level. The parsing level aims to generate a source code model which records all objects, classes and methods interaction of the subject system. Furthermore, the parsing level generates a library of design patterns that has the form of source class, destination class and relation type for all GoF design patterns. On the other hand, the searching level introduces the so-called structural search model (SSM) which involves a searching algorithm for each design pattern. The searching algorithm tries to build the pattern structure incrementally, based on the generated source code model. The third level of MLDA uses a CLIPS inference engine to match the method signatures of the candidate design instances to that of the subject system. Level four introduces a metrics-based approach to assess the impact of design patterns on software maintainability and understandability. This approach relies on software metrics and design pattern occurrences.

Chapter Four

Structural Search Model Evaluation

To evaluate the effectiveness of MLDA in recovering the instances of design patterns, it has been applied to eight open source software systems that are widely used as benchmarks for design pattern recovery. This chapter presents the experimental results of recovering instances of design patterns using the structural search model.

4.1 Introduction

Each design pattern has its own structure which requires a certain arrangement of classes. Each arrangement is connected using two or more relationships. This level of structural information can be used as a starting point to recover instances of design patterns. More specifically, recovering instances of design patterns should search for all possible structures similar to that of GoF. Existing recovery approaches, as presented in Chapter Two, use different searching techniques and library representations to recover instances of design patterns. However, the recovery process should try all the possible structures that are similar to that of GoF design patterns.

This chapter presents the experimental results of applying the structural search model (SSM). The SSM tries to build the structure of each design pattern incrementally, based on the class-level representation of a subject system. The MLDA parser will generate the required class-level representation and store it in an SQL table. The SSM involves a searching algorithm for each design pattern, which tries all the possible combinations between classes until a pattern structure has been found. The classes' combination process relies on the five key relationships. All classes that participate and play a role in the matched structure will be exported into an SQL table. The SSM and its Pseudocode are presented in Appendix B and Appendix C respectively.

The experiments in this chapter aim to address whether the SSM is able to recover instances of design patterns with reasonable detection accuracy. The accuracy in terms of precision and recall and the efficiency in terms of searching time will be used to evaluate the SSM.

4.2 Experiments Setup

MLDA was implemented in Java using NetBeans Integrated Development Environment version 8.1. The recovered instances of design patterns are stored in tables, constructed using MySQL Workbench version 6.3 CE. These systems are open source and implement design patterns in their source codes. Moreover, the selection of these systems was made to facilitate comparison with other existing approaches. All the experiments have been run on Windows 7 with Intel Core i5-2400 CPU.

4.2.1 Subject Systems

MLDA has been applied on JHotDraw, JRefactory, JUnit, QuickUML, Lexi, MapperXML, Nutch and PMD. The selection of these subject systems was made based on their results which are detailed enough to compare. The recovered design pattern instances will be validated based on all publicly published results in the available literature.

JHotDraw is a Java GUI framework for technical and structured graphics. It has been developed as a "design exercise". The design of JHotDraw relies heavily on some well-known design patterns. JHotDraw is a well-designed and flexible framework [87].

JRefactory is a software tool that allows the user to perform different refactoring activities, such as move class between packages, remove empty class, move method and rename parameter. JRefactory tool has the powerful feature of being able to insert the appropriate “Javadoc” comments so that the “Javadoc” program does not generate error messages for missing fields [88].

JUnit is a unit testing framework for the Java programming language. JUnit plays a key role in the development of test-driven development. JUnit is linked as a "JAR" file at compile-time. It resides under the package “junit.framework” for JUnit 3.8 and earlier and under the package “org.junit” for JUnit 4 and later [89].

QuickUML is a UML design tool that supports a highly integrated core set of UML models. It contains advanced features for multiple language projects, design namespaces, UML stereotype extensions, flexible color support, custom detail fields and automated generation of class models from the dictionary [90].

Lexi is a Java-based word processor. It currently edits plain text and RTF files, with HTML and Open Document Format support planned [91].

MapperXML is a presentation framework for web applications. Its framework uses components to build applications. The components follow the Model-View-Controller pattern. MapperXML is extensible for other presentation applications (reporting, data exchange etc) by extending and implementing appropriate containers, components and sub-components [92].

Nutch is a ready web crawler. It enables fine-grained configuration, relying on “Apache Hadoop” data structures which are used for batch processing. In addition, Nutch provides extensible interfaces, such as Parse, Index and Scoring Filters, for custom implementations. Nutch is scalable and robust and can run on a cluster of up to 100 machines. It allows developers to create plug-ins for media-type parsing, data retrieval, querying and clustering [93].

PMD is a source code analyzer. It finds common programming flaws, such as unused variables, empty catch blocks, duplicate code and unnecessary object creation. PMD includes a set of built-in rules and supports the ability to write custom rules [94].

The characteristics of the eight subject systems appear in Table 4.1.

Project	Category	Version	Files	Size (MB)
JHotDraw	Graphics User Interface	5.1	155	2.98
JRefactory	Graphics User Interface	2.6.24	549	4.0
JUnit	Unit Testing	3.7	71	2.66
QuickUML	Design Tool	2001	150	1.76
Lexi	Text Processing	0.1.1 alpha	24	0.84
MapperXML	Presentation Framework	1.9.7	217	2.5
Nutch	Web Crawler	0.4	165	3.0
PMD	Code Analyzer	1.8	446	7.0

Table 4.1: The characteristics of the systems used in the experiments

4.2.2 Effectiveness Evaluation

The effectiveness of MLDA has been evaluated in terms of accuracy and searching time. To evaluate the accuracy, two well-known metrics are used, namely precision and recall. The F-measure, which represents the harmonic mean of recall and precision, is calculated as well. The previous metrics can be calculated as follows [52]:

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \%$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \%$$

$$F - measure = 2 * \frac{Precision * Recall}{Precision + Recall} \%$$

Where:

True Positives are the number of instances correctly detected by MLDA;

False Positives are the number of instances incorrectly detected by MLDA;

False Negatives are the number of instances incorrectly rejected by MLDA (missed instances).

4.2.3 Results Validation

To validate the number of true positives, false positives and false negatives, we refer to all publicly published results in the available literature. In fact, we investigated the results of [31], [33], [37], [41], [42] and [43]. In addition, we used the repository of Perceron [34], the design pattern detection tools benchmark platform [75] and P-MARt [76] as the main benchmarks to validate our results. In doing this, more accurate validation will be conducted. Consequently, after the recovery of all design pattern instances and comparing them with all public results in the available literature, all classes that are playing roles and participating in GoF design patterns can be identified. Moreover, a reference benchmark for all design pattern instances in the investigated subject systems can be generated.

4.3 Recovering Design Pattern Instances

The subject systems have been parsed by the MLDA parser and the SSM model is applied to the generated source code model in attempts to recover the candidate instances of design patterns.

4.3.1 Parsing and Source Code Model Generation

The results of the parsing of the subject systems are presented in Table 4.2. MLDA recovered, in total, 2406 classes and 176 interfaces from the subject systems. The parsing level of MLDA represents each subject system as a set of classes and interfaces and the relationships between them. The MLDA parser exports the generated source code model into an SQL table which has three columns: source class, destination class and relationship type. This SQL table will be examined by SSM to recover instances of design patterns. All relationships have been recovered based on the syntax presented in Table 3.2.

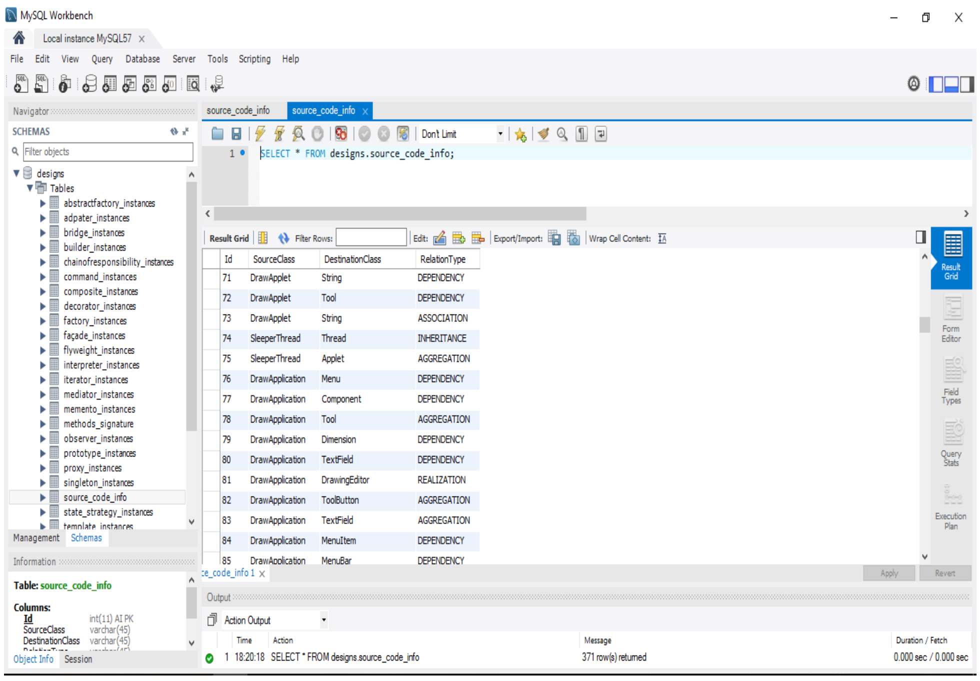
MLDA made a distinction between the inheritance relationship and the realization relationship, in which the realization relationship is an inheritance relationship that has an interface super class. Moreover, MLDA records the object creations inside each class/interface, henceforth class, by distinguishing between the aggregation relationship and the association relationship. Hence, the static analysis conducted by MLDA is acting as a dynamic analysis, since all objects created at the compile time and runtime were recorded.

Recovered features/ Subject Systems	**Classes**	**Interfaces**	**Realizations**	**Inheritances**	**Dependencies**	**Aggregations**	**Associations**	**Parsing Time (seconds)**
JHotDraw	183	18	25	97	110	96	40	39
JRefactory	577	35	89	535	782	439	54	135
JUnit	104	8	13	50	110	31	25	17
QuickUML	126	19	33	105	157	99	118	40
Lexi	151	19	45	64	80	62	43	21
MapperXML	346	28	44	220	175	227	171	55
Nutch	374	24	70	266	455	449	191	40
PMD	545	25	75	464	436	343	155	150
Total	**2406**	**176**	**394**	**1801**	**2305**	**1746**	**797**	**497**

Table 4.2: The results of the parsing of the subject systems using MLDA

As Table 4.2 illustrates, MLDA recovered all the possible relationships that may occur between any two classes inside the Java source code. MLDA is quite fast in parsing the subject systems, taking 497 seconds to parse 1777 files. However, MLDA spent most of the time parsing PMD since it is the largest subject system. This parsing time depends on the size of the subject system and the number of implemented classes. The dependency relationship is the main building relationship for all subject systems. In contrast, the realization relationship is the least implemented relationship.

The number of association and aggregation relationships gives an indication of whether the behavioral design patterns are implemented or are not inside the subject system. MLDA will try to make all the possible arrangements between classes and interfaces to form a pattern structure consistent with the GoF structure presented in the generated library. To explain how MLDA represents the subject systems, a screenshot of the generated source code model of JHotDraw is presented in Figure 4.1.



```
SELECT * FROM designs.source_code_info;
```

Id	SourceClass	DestinationClass	RelationType
71	DrawApplet	String	DEPENDENCY
72	DrawApplet	Tool	DEPENDENCY
73	DrawApplet	String	ASSOCIATION
74	SleeperThread	Thread	INHERITANCE
75	SleeperThread	Applet	AGGREGATION
76	DrawApplication	Menu	DEPENDENCY
77	DrawApplication	Component	DEPENDENCY
78	DrawApplication	Tool	AGGREGATION
79	DrawApplication	Dimension	DEPENDENCY
80	DrawApplication	TextField	DEPENDENCY
81	DrawApplication	DrawingEditor	REALIZATION
82	DrawApplication	ToolButton	AGGREGATION
83	DrawApplication	TextField	AGGREGATION
84	DrawApplication	MenuItem	DEPENDENCY
85	DrawApplication	MenuBar	DEPENDENCY

	Time	Action	Message	Duration / Fetch
1	18:20:18	SELECT * FROM designs.source_code_info	371 row(s) returned	0.000 sec / 0.000 sec

Figure 4.1: A screenshot for the source code model of JHotDraw generated by MLDA

The generated source code model reflects the static behavior of the subject system. Other information can be recovered from the source code, such as abstract classes, concretes classes and fields. However, since SSM relies on the classes arrangements, only the classes and their relationships were recovered.

The SSM will examine each record of the source code model, try to find the required relationships for each pattern and try to form a complete pattern structure.

All the searching algorithms are working in the same manner. The ad-hoc nature of GoF design patterns makes SSM suitable enough for recovering their instances (i.e. the structure and behavior of GoF design patterns has not changed since their introduction in 1995). The SSM was constructed using a series of IF-THEN and FOR statements in such a way that all the possible arrangements of classes and interfaces could be checked. Once a pattern structure is encountered, all pattern participant classes are exported into the corresponding design instances table. The export process is done using an SQL INSERT statement. The inserting process is quite fast, MLDA spending a few seconds inserting the participant classes into the corresponding tables. Each design pattern instances are stored in a separate tables to ease the validation process.

The search time spent by SSM depends on the number of classes involved and on the structure of each design pattern. SSM will search for a complete pattern structure inside the source code model of the subject system. However, some design patterns may be partially implemented in the subject system and considered as a complete design pattern instance. These instances increase the number of false negative instances and are considered as missed instances. One possible solution to allow SSM to recover the instances partially implemented is the exporting of pattern participant classes when a certain number of required relationships is encountered. However, there is no common agreement in the literature on the required number of relationships to recover the partially implemented instances. This motivates us to focus on the standard complete structure of design patterns presented by GoF.

The Pseudocode of the SSM for the Proxy design pattern is presented in Figure 4.2 which shows that all the possible arrangements of classes will be checked until a complete Proxy structure can be formed.

MLDA will try all possible combinations of classes and interfaces until a complete Proxy structure can be formed. The order in which the search attempts are performed is not important since this will not affect the detection accuracy.

```
FOR each record in database {// 1st FOR
    IF (Connecting _Relationship == Realization) {// 1st IF
        s1 = getSourceClass      d1= getDestinationClass
        FOR each record in database {// 2nd FOR
            IF (Connecting _Relationship == Realization){ //2nd IF
                s2 = getSourceClass      d2= getDestinationClass
                IF( s1 != s2 AND  d1==d2) {//3rd IF
                    FOR each record in database  {// 3rd FOR
                        IF (Connecting _Relationship= = association)  {//4th IF
                            s3 = getSourceClass    d3= getDestinationClass
                            IF(s3==s1 AND  d3==s2)  {//5th IF
                                INSERT   d1 into Subject
                                INSERT   s1 into Proxy
                                INSERT   s2 into RealSubject
                            } // end of 5th IF
                        }// end of 4th IF
                    }// end of 3rd FOR
                }// end of 3rd IF
            }// end of 2nd IF
        }// end of 2nd FOR
    }// end of 1st IF
}// end of 1st FOR
```

Figure 4.2: The Pseudocode of the SSM for the Proxy design pattern

4.3.2 Recovering Accuracy

Tables 4.3 and 4.4 present the detailed experimental results of recovering 23 GoF design patterns from the subject systems. An instance of a design pattern is said to be a candidate instance if it has a structure similar to that of GoF. As the experimental results illustrate, the SSM is performing quite well, recovering 2994 candidate instances within 551 seconds. However, SSM spent a longer time recovering the instances of the Memento and Abstract Factory design patterns since they have more complicated structures than do other design patterns.

In terms of accuracy, SSM detected most of the instances that are consistent with standard structural definitions presented by GoF. However, some instances have a structure similar to that of the GoF design patterns yet are not design patterns. These instances increase the number of false positive instances and affect SSM's accuracy. As the number of required relationships for each instance increases, the chances of that instance being a true positive instance increases.

SSM missed only those instances that are partially implemented in the source code. For example, it rejected one Visitor instance since the roles of "ObjectStructure", "Element" and "ConcreteVisitor" were not implemented in the source code of JRefactory. The partly implemented instances affect the recall rate of SSM which is unable to distinguish between the State and Strategy design patterns since both have a similar structure and require dynamic analysis capabilities to distinguish between their instances.

The most commonly implemented design patterns are Singleton and Façade which are implemented in all subject systems. In general, most software systems rely heavily on Singleton and Façade design patterns since their intents help these systems to fulfill their functionality. In contrast, the Interpreter design pattern is not implemented in any of the subject systems.

As the experimental results show, the relationships matching is not enough to detect GoF design patterns. Although all five major relationships are recovered from the subject systems, the structure of the candidate instances results in too many false positive instances. However, SSM detects all the Singleton instances correctly and only two instances are missed in PMD. Furthermore, SSM is unable to detect instances of the Template design pattern since its structure relies only on the inheritance relationship (SSM will search for two participant classes, Abstract and Concrete, that are connected together using an inheritance relationship). Hence, too many false positive instances will be detected.

In some cases, where the number of implemented design pattern instances in a subject system is zero, the corresponding precision cannot be calculated and its value is set to NA (Not Applicable). In addition, the recall is set to NA when there is no benchmark reports the correct instances in a subject system. Hence, the number of missed instances cannot be determined. However, this case only happened when we tried to validate JRefactory instances.

Subject Systems/ Design Patterns	JHotDraw							JRefactory							JUnit							QuickUML						
	CI	**TP**	**FP**	**FN**	**P%**	**R%**	**T**	**CI**	**TP**	**FP**	**FN**	**P%**	**R%**	**T**	**CI**	**TP**	**FP**	**FN**	**P%**	**R%**	**T**	**CI**	**TP**	**FP**	**FN**	**P%**	**R%**	**T**
Singleton	2	2	0	0	100	100	0	10	10	0	2	100	83	2	0	0	0	0	NA	NA	0	1	1	0	0	100	100	1
Prototype	2	2	0	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	1	0	0	0	0	NA	NA	3
Abstract Factory	0	0	0	0	NA	NA	1	0	0	0	0	NA	NA	20	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	1
Factory Method	0	0	0	2	NA	0	0	101	79	22	8	78	91	21	2	2	0	0	100	100	1	12	12	0	6	100	67	4
Builder	0	0	0	0	NA	NA	0	0	0	0	2	NA	0	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	0
Adapter	31	11	20	0	35	100	1	17	15	2	1	88	94	1	7	5	2	6	71	45	1	26	25	1	4	96	86	1
Bridge	7	4	3	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	2
Composite	1	1	0	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	1	1	1	0	0	100	100	1
Decorator	1	1	0	2	33	33	0	1	1	0	0	100	100	0	1	1	0	0	100	100	0	2	2	0	1	100	67	0
Façade	1	1	0	0	100	100	0	2	2	0	NA	100	NA	1	0	0	0	0	NA	NA	0	1	1	0	0	100	100	0
Flyweight	1	1	0	0	100	100	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	1	1	0	0	100	100	0
Proxy	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	1	1	0	1	100	50	0
CoR	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0
Command	17	8	9	1	47	89	2	45	22	23	3	49	88	15	0	0	0	0	NA	NA	0	17	17	0	1	100	94	1
Interpreter	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0
Iterator	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	2	0	0	0	1	NA	0	0	0	0	0	0	NA	NA	0
Mediator	2	0	2	0	0	NA	0	0	0	0	NA	NA	NA	18	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0
Memento	1	0	1	0	0	NA	8	0	0	0	NA	NA	NA	9	0	0	0	0	NA	NA	3	6	0	6	0	0	NA	4
Observer	0	0	0	2	NA	0	0	0	0	0	0	NA	NA	2	0	0	0	1	NA	0	0	1	1	0	0	100	100	1
State/Strategy	33	6	27	0	18	100	3	11	8	3	3	73	73	4	8	3	5	0	38	100	1	11	10	1	0	91	100	0
Visitor	1	1	0	1	100	50	1	1	1	0	1	100	50	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0
Template Method	95	4	91	0	4	100	4	535	4	531	0	1	100	13	50	1	49	0	2	100	3	105	3	102	0	3	100	5
Total/Average	195	42	153	8	22%	84%	20	723	142	581	20	0.2%	88%	108	68	12	56	9	18%	57%	11	185	75	110	16	41%	82%	24

Note:
CI: Candidate Instances after applying Structural Search Model (level two)
P: Precision% **R**: Recall% **NA**: Not Applicable
TP: True Positives **FP**: False Positives **FN**: False Negatives
T: searching time (seconds)

Table 4.3: The experimental results of recovering 23 GoF design patterns from the subject systems-part 1

Subject Systems/ Design Patterns	Lexi							MapperXML							Nutch							PMD						
	CI	TP	FP	FN	P%	R%	T	CI	TP	FP	FN	P	R	T	CI	TP	FP	FN	P%	R%	T	CI	TP	FP	FN	P%	R%	T
Singleton	2	2	0	0	100	100	0.6	3	3	0	0	100	100	0.4	2	2	0	0	100	100	0.4	2	2	0	2	100	50	0.4
Prototype	1	1	0	0	100	100	0.4	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.4	1	1	0	0	100	100	0.5
Abstract Factory	0	0	0	0	NA	NA	0.8	0	0	0	2	NA	0	3.1	0	0	0	0	NA	NA	48.4	0	0	0	2	NA	0	18.2
Factory Method	26	0	26	0	NA	NA	1.7	0	0	0	22	NA	0	2.6	8	5	3	0	63	100	11.0	7	7	0	25	100	22	15.0
Builder	13	0	13	0	NA	NA	1.0	2	1	1	0	50	100	0.5	0	0	0	0	NA	NA	0.4	16	16	0	0	100	100	1.3
Adapter	44	9	35	21	20	30	2.1	17	5	12	36	29	12	1.3	63	44	19	0	70	100	4.0	53	42	11	45	79	48	2.5
Bridge	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.4	0	0	0	0	NA	NA	0.6	1	1	0	0	100	100	0.6
Composite	3	3	0	0	100	100	0.5	0	0	0	1	NA	0	0.3	0	0	0	0	NA	NA	0.4	0	0	0	0	NA	NA	0.4
Decorator	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.4	0	0	0	4	NA	0	0.5	0	0	0	0	NA	NA	0.5
Façade	1	0	1	0	NA	NA	0.4	1	1	0	0	100	100	0.7	1	1	0	0	100	100	1.6	1	1	0	0	100	100	0.9
Flyweight	1	1	0	0	100	100	0.4	1	1	0	0	100	100	0.5	1	1	0	0	100	100	0.6	0	0	0	0	NA	NA	0.5
Proxy	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.4	4	4	0	0	100	100	0.9	3	3	0	1	100	75	0.8
CoR	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.3	1	1	0	0	100	100	0.4
Command	15	6	9	0	40	100	0.9	52	26	26	0	50	100	13.3	23	23	0	2	100	92	22.0	3	3	0	0	100	100	13.9
Interpreter	0	0	0	0	NA	NA	0.3	0	0	0	0	NA	NA	0.4	0	0	0	0	NA	NA	0.5	0	0	0	0	NA	NA	0.5
Iterator	36	0	36	0	NA	NA	4.6	0	0	0	0	NA	NA	0.6	0	0	0	1	NA	0	1.8	0	0	0	0	NA	NA	2.1
Mediator	0	0	0	0	NA	NA	0.7	0	0	0	0	NA	NA	2.2	0	0	0	0	NA	NA	8.4	0	0	0	0	NA	NA	14.4
Memento	10	0	10	0	NA	NA	1.0	12	12	0	0	100	100	9.9	1	1	0	0	100	100	37.8	1	1	0	0	100	100	24.3
Observer	0	0	0	0	NA	NA	0.7	0	0	0	3	NA	0	2.2	0	0	0	0	NA	NA	7.3	0	0	0	1	NA	0	12.7
State/Strategy	0	0	0	1	NA	0	0.3	11	11	0	0	100	100	0.8	124	113	11	0	91	100	6.0	248	21	227	0	8	100	10.3
Visitor	0	0	0	0	NA	NA	0.4	0	0	0	0	NA	NA	0.8	0	0	0	0	NA	NA	8.8	0	0	0	1	NA	0	0.5
Template Method	64	0	64	0	NA	NA	2.7	220	56	164	0	79	100	9.1	260	7	253	0	3	100	13.3	464	108	356	0	23	100	20.5
Total/Average	194	22	194	22	10%	50%	21	319	116	203	64	36%	64%	51	487	201	286	7	41%	97%	175	801	207	594	77	26%	73%	141

Note:
CI: Candidate Instances after applying Structural Search Model (level two)
P: Precision% **R**: Recall% **NA**: Not Applicable
TP: True Positives **FP**: False Positives **FN**: False Negatives
T: searching Time (seconds)

Table 4.4: The experimental results of recovering 23 GoF design patterns from the subject systems-part 2

Table 4.5 presents the average precision, recall and F-measure of SSM for recovering all GoF design patterns and for recovering all GoF design patterns excluding the Template Method design pattern.

	All GoF design patterns					**All GoF design patterns except Template Method design pattern**			
Subject Systems	CI	TP	FP	FN		CI	TP	FP	FN
JHotDraw	195	42	153	8		100	38	62	8
JRefactory	723	142	581	20		188	138	50	20
JUnit	68	12	56	9		18	11	7	9
QuickUML	185	75	110	16		80	72	8	16
Lexi	216	22	194	22		152	22	130	22
MapperXML	319	116	203	64		99	60	39	64
Nutch	487	201	286	7		227	194	33	7
PMD	801	207	594	77		337	99	238	77
Total	2994	817	2177	223		1201	634	567	223
Average precision	27%					53%			
Average recall	79%					74%			
Average F-measure	41%					62%			

Note:
CI: Candidate Instances after applying Structural Search Model (level two)
TP: True Positives **FP**: False Positives **FN**: False Negatives

Table 4.5: The average accuracy of SSM

The accuracy of SSM increased when excluding the Template Method design pattern from the pattern detection list. However, the accuracy is still not reasonable since too many false positive instances are recovered. SSM achieved its highest precision when recovering instances of QuickUML. In contrast, SSM shows the lowest accuracy when recovering instances of JRefactory. This is mainly due to the instances' nature inside these systems. Most JRefactory instances are partly implemented and have a structure similar to that of GoF, yet they are not GoF. SSM achieved an average recall of 79% for all GoF design patterns, which indicates the ability of SSM to recover most of the instances that have a complete GoF structure.

The structure of a design pattern is not enough to detect its instances from the source code. To enhance the detection process, which relies on the relationships matching, the third level of MLDA has been developed. The new level is aimed at reducing the number of false positive instances (i.e.

the instances that have a structure similar to that of GoF, but are not design patterns).

Design patterns are not only about the structure and the specific arrangement of classes. GoF illustrates that a design pattern should implement certain method signatures, such as whether a method is static or not, method return type and method access modifier. Furthermore, some pattern participant classes should implement one or more methods that call a method, implemented in another participant class. The third level of MLDA tries to match the required method signatures of the candidate design pattern instances to that of the subject system to reduce the number of false positive instances. In the next two chapters, a rule-based approach will be presented and evaluated to match the method signatures of the candidate design instances generated by SSM.

4.4 Results Comparison

The accuracy of the Structural Search Model (SSM) has been compared to four approaches, as presented in Tables 4.6 and 4.7. The selection of these approaches was made based on their results which were detailed enough to compare and were applied to the same subject systems (JHotDraw version 5.1 and JUnit version 3.7). However, the comparison among design pattern detection approaches is challenging. This is due to the fact that there is no standard benchmark to validate the results of each approach. In fact, each approach has its limitations, patterns representation, subject systems and validation method. Tables 4.6 and 4.7 show the results of the design patterns recovery of SSM, Sempatrec [44], DeMIMA [41], Sub-patterns [33] and SSA [31] for JHotDraw and JUnit respectively. However, SSM missed the instances partly implemented in the source code, since SSM relies on the standard definition of GoF. On the other hand, the lack of dynamic and runtime information explains the existence of false positives. It must be noted that we only compare the results that DeMIMA, SSA, Sempatrec, and Sub-patterns revealed.

DPs	SS	SSM			DeMIMA			SSA			Sempatrec			Sub-patterns		
		P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
AD	JD	35	100	52	4	100	8	44	100	61	45			100	NA	NA
DE	JD	100	33	50	8	100	15	33	33	33	50	33	40	100	NA	NA
CO	JD	100	100	100	33	100	50	100	100	100	100	100	100	100	NA	NA
FM	JD	NA	0	NA	2	100	4	100	67	80	100	100	100	100	NA	NA
SI	JD	100	100	100	100	100	100	100	100	100	100	100	100	100	NA	NA
OB	JD	NA	0	NA	25	100	40	50	40	44	50	40	44	100	NA	NA
TM	JD	4	100	8	7	100	13	20	100	33	50	100	67	100	NA	NA
VI	JD	100	50	67		100		100	100	100				100	NA	NA
Average %		**73**	**60**	**63**	**26**	**100**	**33**	**68**	**80**	**69**	**71**	**79**	**75**	**100**	**NA**	**NA**

Note:
AD: Adapter DE: Decorator CO: Command FM: Factory Method SI: Singleton OB: Observer TM: Template Method
VI: Visitor SS: Subject Systems JD: JHotDraw JU: JUnit P:Prescion R: Recall F: F-measure
Blank: Not revealed NA: Not Applicable since the number of detected instances is zero or there is no reference to validate the instances

Table 4.6: Comparison of the results of SSM and that of other approaches for JHotDraw

DPs	SS	SSM			DeMIMA			SSA			Sempatrec			Sub-patterns		
		P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
AD	JU	71	45	55	0			17	100	29	100	100	100	100	100	100
DE	JU	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
CO	JU	NA	0	NA	100	100	100	100	100	100	100	100	100	100	100	100
FM	JU	100	100	100		100								100	100	100
SI	JU	NA	NA	NA										100	100	100
OB	JU	NA	0	NA	25	100	40	100	100	100	100	100	100	100	50	67
TM	JU	2	100	4	0			100	100	100	100	100	100	100	100	100
VI	JU	NA	NA	NA		100									100	100
Average %		**68**	**58**	**65**	**45**	**100**	**80**	**83**	**100**	**86**	**100**	**100**	**100**	**100**	**94**	**96**

Note:
AD: Adapter DE: Decorator CO: Command FM: Factory Method SI: Singleton OB: Observer TM: Template Method
VI: Visitor SS: Subject Systems JD: JHotDraw JU: JUnit P:Prescion R: Recall F: F-measure
Blank: Not Revealed NA: Not Applicable since the number of detected instances is zero or there is no reference to validate the instances

Table 4.7: Comparison of the results of SSM and that of other approaches for JUnit

4.5 Threats to Validity

Threats to internal validity concern factors that could affect the results. In this dissertation, this is mainly due to the variants of design patterns. Design pattern instances are recovered based on the standard structural format presented by GoF [11]. Moreover, the way in which the results are validated could affect precision and recall. To validate the number of true positives, false positives and false negatives, we refer to all publicly published results in the available literature. In fact, we investigated the results of [31], [33], [37], [41], [42] and [43]. In addition, we used the repository of Perceron [34], the design pattern detection tools benchmark platform [75] and P-MARt [76] as the main benchmarks for validating our results. In doing this, a more accurate validation is performed. In total, we validated 2994 candidate instances based on all public results presented by other approaches.

Threats to external validity concern the generalization of the results. In fact, this thesis focuses on Java programming language. It could be worthwhile to conduct the evaluation on other projects having different languages.

4.6 Summary

This chapter presented the experimental results of recovering design pattern instances from eight subject systems using the structural search model. Specifically, the parsing level and the searching levels of MLDA are applied on JHotDraw, JRefactory, JUnit, QuickUML, Lexi, MapperXML, Nutch and PMD.

The Parsing level recovered the five key relationships that may occur between all classes and interfaces inside any object-oriented program. The MLDA parser is quite fast, recovering 2582 classes and interfaces within 497 seconds. The output of the parsing level is a source code model that has the form of source class, destination class and relationship type.

SSM has been applied to the generated source code model to recover the design pattern instances that have a complete GoF structure. SSM cannot recover instances that are partly implemented in the source code.

As the experimental results illustrate, the relationship matching is not enough to recover the instances of GoF design patterns. SSM produces too many false positive instances since too many instances have a structure similar to that of GoF design patterns, but they are not design patterns.

However, SSM achieved an average recall of 79% since it only missed the partially implemented instances.

Chapter Five

Method Signatures Matching

The Structural Search Model (SSM) of MLDA detected the instances of design patterns based on the connecting relationships between pattern participant classes. However, the structure of each design pattern is not enough to detect all the instances correctly and produces too many false positives. This chapter presents the third level of MLDA, which applies a rule-based approach to enhance the detection accuracy achieved by SSM.

MLDA applies a rule-based system to filter the candidate design pattern instances detected by SSM. The rule-based system tries to match the method signatures of the candidate design pattern instances to that of the subject system. A rules template for the method signatures of GoF design patterns has been created. Specifically, the method signatures of the candidate design pattern instances were represented as a list of rules. Based on the rules template, the MLDA rules/facts generator will generate a list of rules to reflect the required method signatures and method calls between participant classes of the candidate instances. On the other hand, the MLDA rules/facts generator will generate a list of facts to represent the interactions between methods inside the subject system. MLDA uses CLIPS, an expert system tool, to process the generated facts and rules and to remove the false positive instances. The generated rules are consistent with the required method signatures presented by GoF [11].

5.1 Rule-Based Systems

Rule-based systems use expert knowledge to solve real-world problems that would normally require human intelligence. A rule-based system contains IF-THEN rules, facts and an inference engine that controls the application of the rules. Specifically, rule-based systems represent knowledge in terms of a bunch of rules that inform the knowledge engineer what to conclude in different situations. The term “rule” can be defined as an IF-THEN structure that relates given facts in the IF part to some action in the THEN part. An important advantage of rule-based systems is that, within the domain of the knowledge base, a different problem can be solved using the same program without reprogramming efforts. Rules can represent relations, recommendations, directives, strategies and heuristics. When the condition part of a rule is satisfied, the rule is said to fire and the action part is executed.

A typical rule-based system consists of the following components [95]:

- A knowledge base which contains the rules that represent expert knowledge about the problem domain. Knowledge acquisition is a key element in the development of expert systems. Knowledge could be obtained by learning and experience.
- The database, a working memory, which contains the set of known facts about the problem used to match against the IF (condition) parts of rules stored in the knowledge base.
- An inference engine which links (compares) rules stored in the knowledge base with known facts provided in the database to reach a conclusion. The matching of the rule IF parts to the facts produces inference chains. The inference chain indicates how the expert system applies the rules to reach a conclusion.
- Explanation facilities which provide information to the user about the reasoning steps.
- A user interface which allows interactions between the user and the system.

The architecture of a rule-based system appears in Figure 5.1:

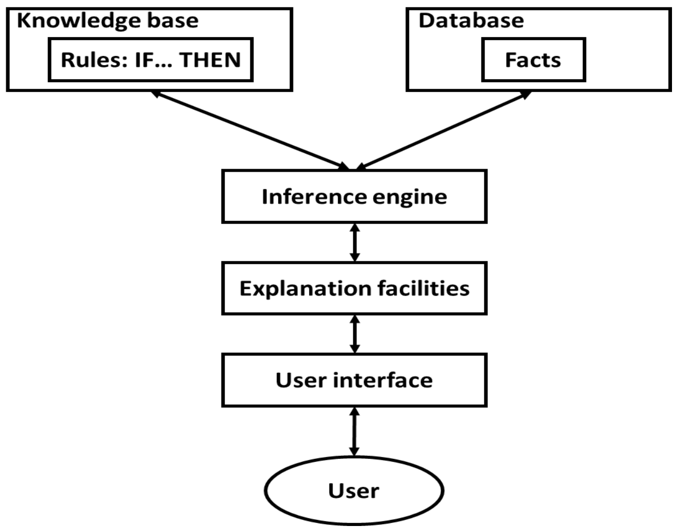


Figure 5.1: The basic architecture of a rule-based system

Rule-based systems have high-quality performance, employ symbolic reasoning when solving a problem and apply a heuristic to guide the reasoning. Thus, the search area for a solution is reduced. There are two kinds of inference engines used in rule-based systems: forward chaining and backward chaining.

In forward chaining systems, the initial facts are processed first and rules are used to draw new conclusions given those facts. Forward chaining is useful when no specific goal is being explored. It is appropriate in situations where data are expensive to collect, but few in quantity. Furthermore, forward chaining is data-driven in that each time only the topmost rule is executed. Any rule can be executed only once. The match-fire cycle stops when no further rules can be fired. However, forward chaining would not be efficient when the goal is to infer only one particular fact (if the system needs to gather some information and then tries to infer from them whatever can be inferred, forward chaining is the best choice).

In backward chaining systems, the hypothesis (goal, solution) is processed first and keeps looking for the rules that would allow a conclusion to the hypothesis. It is goal-driven reasoning. Backward chaining is useful in situations where the quantity of data is potentially very large and where some specific characteristics of the system under consideration are of interest. If the system begins with a hypothetical solution and then attempts to find facts to prove it, backward chaining is the appropriate choice. The same set of rules can be used for both forward and backward chaining.

One of the key concepts in rule-based systems is conflict resolution. Conflict resolution can be defined as the method of choosing a rule to fire when more than one rule can be fired in a given cycle. Some approaches are used to handle conflict resolution, such as:

- Firing the rule with the highest priority. The priority can be established by placing the rules in an appropriate order in the knowledge base. Hence, the topmost rule will be fired first.
- Firing the most specific rule (also known as the longest matching strategy). This is based on the assumption that a specific rule processes information more than a general one does.
- Firing the rule that uses the data most recently entered in the database. The inference engine first fires the rules whose condition uses the data most recently added to the database. Tags can be attached to each fact in the database.

Rule-based systems are developed using specialized software tools called shells. These shells come equipped with an inference mechanism (forward chaining, backward chaining, or both). The shell provides the key components of a rule-based system. The knowledge is entered based on a specific format. Examples of shells are JESS, CLIPS, Drools, rules engine and e2glite. The inference process used in a rule-based system is deductive inference. This means that the rules of logic are used to deduce new knowledge from existing rules and knowledge. CLIPS is one of the most popular shells, widely used through the industry and in academia [78]. CLIPS is written in C and supports three programming paradigms: object-oriented, rule-based and procedural. CLIPS uses forward chaining and provides a language for representing facts and rules. The language is based on the artificial intelligence language LISP. CLIPS inference engine does the required matching between facts and rules using the “Rete algorithm” [96].

5.1.1 Motivation

The method signatures of GoF design patterns share common characteristics and features, such as methods overriding, methods call and level of abstraction. Rule-based systems use human experience in a specific field to build an intelligent system that mimics human behavior.

Our manual attempts to match the method signatures of the candidate design pattern instances to that of the subject system produces a set of IF-THEN statements. Rule-based systems provide a separation of knowledge from system processing. Specifically, the structure of the rule-based systems provides an effective separation of the knowledge base from the inference engine.

In fact, our main motivation behind the use of a rule-based system to match the method signatures of the candidate design instances to that of the subject system is the ability to represent the method signatures of the candidate design instances as an independent piece of knowledge, which can be transformed into a set of rules. In addition, the method signatures of GoF design patterns have a uniform structure which facilitates their representation as a set of rules. On the other hand, the comparison process that the inference engine can perform allows the effective matching of the set of rules to the facts. Rule-based systems can deal with uncertain and incomplete knowledge. Specifically, part of the method signatures can be used to generate a set of rules.

However, rule-based systems are unable to learn. Thus, the matching process is trying to match “exactly” the method signatures of the candidate design instances to that of the subject system. In addition, a rule-based system cannot automatically modify its knowledge base, adjust existing rules, or add new ones. In the case of method signatures matching, there is no need to modify the existing rules. All rules will be generated based on the created template of GoF method signatures. The structure and method signatures of GoF design patterns did not change over time since their introduction in 1995.

The use of object-oriented programming languages to implement the matching process between facts and rules leads to too many nested IF-THEN statements, too many loops, and complicated matching structure. This leads to an inefficient, inaccurate and hard-coded matching process.

The use of a rule-based system aims to remove the false positive instances detected by SSM. The fundamental hypothesis that we want to explore is as follows: representing the method signatures of the candidate design instances and subject systems using a rule-based system is effective in improving the precision of recovering design pattern information which relies on relationships matching. CLIPS, an expert system tool, will be used to process the facts and rules that represent the method signatures of the subject system and the candidate design instances respectively. CLIPS supports forward chaining and is widely used in academia and industry.

5.1.2 Rule-Based Systems and Design Patterns

The use of rule-based systems is not widely adopted in the area of design pattern detection. Only two approaches adopted the idea of using rules and method signatures to detect instances of design patterns. The approach presented by Alnusair *et al.* [44], Sempatrec, uses ontology formalism to represent the conceptual knowledge of the source code and semantic rules to capture the structures and behaviors of design patterns in the subject system. Sempatrec presents an ontology model that includes a Source Code Representation Ontology (SCRO) which explicitly represents the conceptual-knowledge structure found in the source code. SCRO captures the key concepts and features of object-oriented programs, such as methods overriding, method signatures and invocations, aggregation between objects and control structures (repetition, sequence controls and selection). SCRO's knowledge is represented using the Web Ontology Language (OWL-DL), a sub-language of OWL, based on Description Logic (DL).

Various object properties and data properties are defined within SCRO to represent the relationship between concepts by linking individuals from different OWL classes. For example, "hasOutputType" is an object-functional property defined for the return type of a method. A system, the knowledge generator, has been built to extract knowledge automatically from Java bytecode. It captures every SCRO concept that represents a source-code element. Moreover, the knowledge generator generates instances for all ontological properties defined in SCRO. The semantic instances generated by the knowledge generator subsystem are serialized using Resource Description Framework (RDF) and linked to SCRO or any other OWL ontology. Figure 5.2 shows part of the knowledge base and an RDF description of JHotDraw generated by the Sempatrec knowledge generator.

```
scro: hasAccessControl scro: public;
scro: hasSuperType
<#org. jhotdraw.util.Storable >;
scro: use <#org. jhotdraw. framework. Connector >;
scro: hasAbstractMethod
<#org. jhotdraw. framework. Figure . draw [. . . .] > ;
. . . .
. . . .
<#org. jhotdraw. framework. Figure. draw [. . . .] >
rdf: type scro: AbstractMethod;
scro: hasOutputType <#void >;
scro: hasInputType <#java. awt. Graphics >;
scro: hasSignature
" org. jhotdraw . framework. Figure.draw [ . . .]";
scro: hasAccessControl scro: public.
. . . .
```

Figure 5.2: Part of Sempatrec's knowledge base representation of JHotDraw

Sempatrec applies a rule-based approach to detect instances of design patterns. Specifically, the OWL-DL inference engine computes the entailments from a set of facts and Semantic Web Rule Language (SWRL) rules defined in the ontologies. The detection process is based on the logical inference that requires a rule-based reasoner capable of processing the SWRL rules.

The inference engine will recover pattern instances based on a matching between the semantic constraints specified in the rules and the source code descriptions found in the knowledge base representing the subject system. Further details on the Sempatrec recovery process were presented in Chapter Two.

The idea of using method signatures matching was adopted by Dongjin *et al.* where they presented a sub-pattern representation for 23 GoF design patterns (a sub-patterns approach) [33]. The recovery process of the sub-patterns approach was presented in detail in Chapter Two. The sub-patterns approach detected the instances of design patterns by the matching sub-graphs of the class-relationships directed graph that represents the subject system. The method signatures of the candidate design instance are analyzed and matched to a predefined template to achieve the final instances. However, it is not clear how the matching process is performed. The sub-patterns approach did not explain how the method signatures of the candidate instances are matched to that of the subject system. We tried to contact the authors, but unfortunately, there was no response.

We used the concepts of method signatures and rule-based systems to filter the candidate design instances and enhance SSM's recovery accuracy. The method signatures provide more concrete behavior of pattern participant classes. The MLDA rules/facts generator has been developed to generate the required rules and facts. On the other hand, the inference engine of CLIPS will be used to match the generated set of rules and facts.

5.2 CLIPS

The 'C' Language Integrated Production System (CLIPS) is a rule-based programming language useful for creating expert systems. It attempts to match the patterns of rules against the facts in the rule list. CLIPS is also useful for creating other programs where a heuristic solution is easier to implement than an algorithmic solution. CLIPS was developed at NASA's Johnson Space Center from 1985 to 1996 and, since then, it has been available as a public domain software [78].

CLIPS has been designed in a way to facilitate the development of software that model human knowledge or experience. CLIPS provides two ways to model the knowledge:

- Rules that are mainly designed to support heuristic knowledge based on experience;

- Deffunctions, generic functions and object-oriented that are mainly intended for procedural knowledge.

The rule-based system can be developed using only rules, only objects, or a mixture of rules and objects. The CLIPS shell provides the basic elements of an expert system:

- A fact-list which contains all the facts about the problem. Facts are stored in short-term memory.
- A knowledge-base which contains all the rules. Rules are stored in the knowledge base (database).
- An inference engine which controls the overall execution of the rules.

In procedural languages, such as C, Ada and BASIC, the execution can proceed without data. In contrast, data are required to cause the execution of rules in CLIPS which is a forward chaining system, starting from the facts to develop a solution. The CLIPS inference engine uses “Rete algorithm” for rules and facts matching.

5.2.1 Rete Algorithm

Rete algorithm is a pattern matching algorithm for implementing expert systems, designed by Charles L Forgy in 1974 [96]. It is used to determine which rule the inference engine should fire. Rete algorithm aims to speed up the pattern matching process. It is a directed acyclic graph that represents higher level rule sets.

A Rete-based expert system builds a network of nodes where each node, except the root node, corresponds to the condition part of the rule. A complete rule left-hand side can be defined by tracing the path from the root node to a leaf node. The new inserted facts are propagated along the network, causing nodes to be annotated when that fact matches the pattern. When a leaf node is reached, this indicates that all the conditions (patterns) for a given rule are satisfied. Thus, the corresponding rule is fired. Rete algorithm stores information about matches in a network structure.

Rete becomes the basis for many expert system shells, including Jess, Drools, BizTalk, Rule engine, Soar and Sparkling Logic SMARTS and CLIPS.

Rete algorithm uses node sharing to reduce a certain type of redundancy. The ability of Rete algorithm to store partial matches allows the production system to avoid complete re-evaluation of all facts each time changes are made. However, Rete algorithm is theoretically independent of the number of rules in the system [96].

When facts are asserted to the working memory, the inference engine creates Working Memory Elements (WMEs) for each fact. Each WME enters the Rete network at a single root node. It may then be propagated through the network until it arrives at a terminal node. Rete algorithm constructs a matching network from the conditions of a set of rules. The inputs of Rete algorithm are a set of facts and the outputs are activation records that indicate how rules match against facts. Rete algorithm avoids redundant matching and shares matching across rules with common conditions.

Figure 5.3 shows a simple example to illustrate how Rete algorithm matches the set of facts against rule conditions. For a single rule's condition, Rete algorithm will construct a single-input node for each fact-value test. The combination of the first two single-input nodes produces a so-called dual-input node which outputs facts that match both tests. Another dual-input node is constructed from the previous dual-input node and the next single-input node. This step is repeated until all tests in the rule's conditions are plugged into the chain of dual-input nodes.

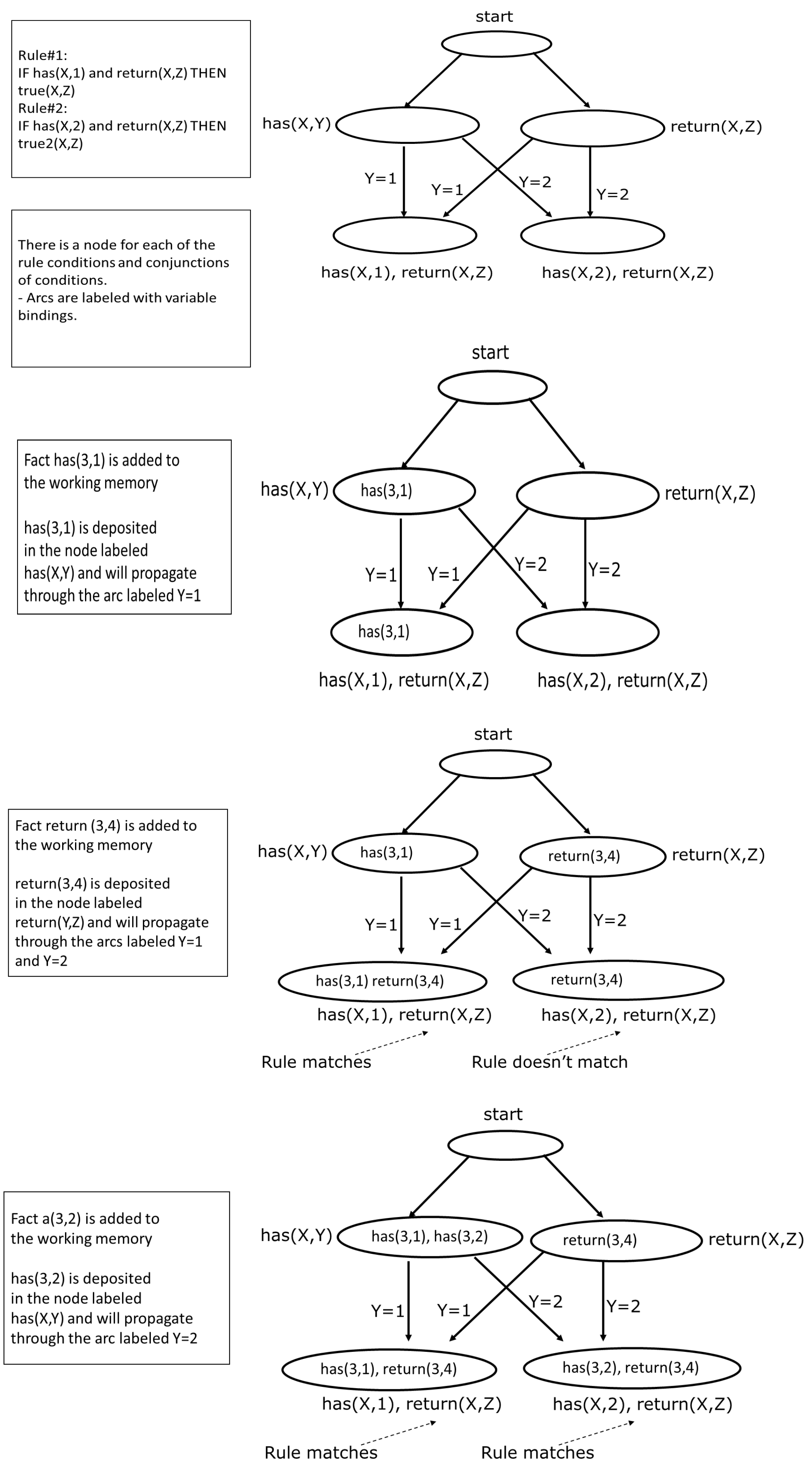
For a second rule condition, Rete algorithm re-uses single-input nodes. If Rete algorithm does not already construct a single-input node for a fact-value test in the second rule's condition, it will construct a new one. Furthermore, Rete algorithm re-uses dual-input nodes when the fact-value tests appear in the same order in the second rule's condition as in the first. However, a different set of dual-input nodes will be constructed if the tests are in a different order in the second rule's conditions.

5.2.2 Rules and Facts Matching in CLIPS

CLIPS attempts to match the patterns (conditions) of rules against the facts in the rule list. When all patterns of a rule match the facts, the rule is activated and put on the agenda (i.e. the action part will be executed when the rule's condition part matches fact(s) in the working memory). Hence, the rule fires. The term "fire" means that CLIPS has selected a certain rule for execution from the agenda.

The agenda is a collection of activations which are rules that match pattern entities. Zero or more activations may be on the agenda.

However, when multiple activations are on the agenda, CLIPS automatically determines which activation is appropriate to fire. The activations are ordered by CLIPS in terms of increasing priority. Specifically, the CLIPS inference engine sorts the activations according to their salience, i.e. the topmost rule will be fired first. This sorting process, as illustrated in the previous section, is called conflict resolution because it eliminates the conflict of deciding which rule should be fired next. The "run" command will fire all rules in the agenda.



Rule#1:
IF has(X,1) and return(X,Z) THEN true(X,Z)
Rule#2:
IF has(X,2) and return(X,Z) THEN true2(X,Z)

There is a node for each of the rule conditions and conjunctions of conditions.
- Arcs are labeled with variable bindings.

Fact has(3,1) is added to the working memory

has(3,1) is deposited in the node labeled has(X,Y) and will propagate through the arc labeled Y=1

Fact return (3,4) is added to the working memory

return(3,4) is deposited in the node labeled return(Y,Z) and will propagate through the arcs labeled Y=1 and Y=2

Fact a(3,2) is added to the working memory

has(3,2) is deposited in the node labeled has(X,Y) and will propagate through the arc labeled Y=2

Figure 5.3: A simple Rete algorithm example

5.3 Methodology

The candidate design pattern instances that have been detected by MLDA's Structural Search Model (SSM) will be filtered by applying a rule-based approach. The rule-based approach aims to remove the false positive instances by matching the method signatures of the candidate design instances to that of the subject system. A rules template for GoF method signatures has been created to reflect the required method signatures for each design pattern. In addition, we introduce what is the so-called MLDA rules/facts generator, a simple Java program that is able to write a set of rules and facts based on the method signatures representation of the candidate design instances and subject system. Figure 5.4 shows the architecture of MLDA's level three.

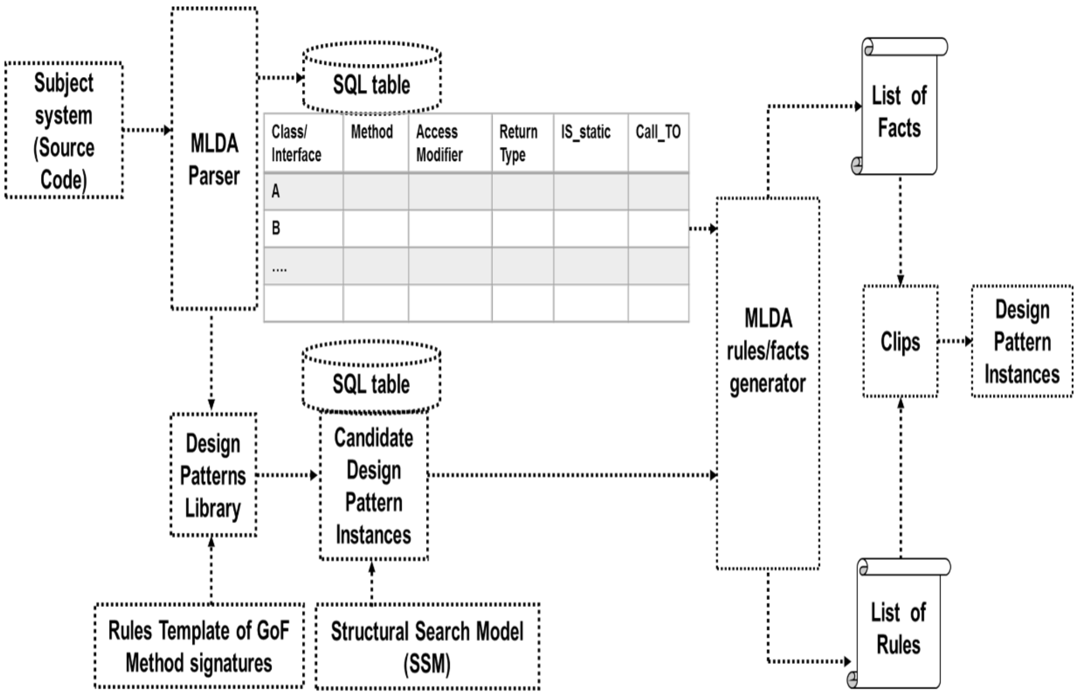


Figure 5.4: The architecture of MLDA's level three

The hybrid system architecture illustrates the relationship between SSM, the MLDA rules/facts generator, the rules template and CLIPS. In fact, this architecture is an enhancement of the initial MLDA architecture presented in Chapter Three. As Figure 5.4 illustrates, the MLDA rules/facts generator make access to two tables: the candidate instances table recovered by SSM and the method signatures of the subject system.

Furthermore, the outputs of the MLDA rules/facts generator are two files (.txt files) that store the generated set of rules and facts. These files will be loaded into CLIPS where its inference engine will do the required processing and the matching between facts and rules. The output of the inference engine is a print message indicating that a candidate instance is a true positive. The hybrid system uses the MLDA parser to parse the subject system and to recover its method signatures from each class/interface. The MLDA parser, as presented in Chapter Three, is developed based on Javaparser version 1.0.11 [80] which generates an Abstract Syntax Tree (AST) representation of the static behavior of the subject system.

The MLDA parser recovers all methods that are implemented inside each class/interface, with the following signatures:

- Access modifier signature which can be public, private, or protected.
- Return type signature which can be void, data type, or another class.
- Is_static signature which indicates whether the method is static or not. This table field is set to YES or NO.
- Call_to signature which indicates if the method makes a call to another method. This table field is kept blank if the method does not call another method.

Consequently, the outputs of the MLDA parser are method signatures, a source code model and a design patterns library. The method signatures template acts as a basis for generating the rules. The template involves a rule template for each design pattern. The template aims to represent the key method characteristics between the pattern participant classes, such as method overriding (superclass and subclass implement the same method), method calls (a method in one class calls a method in another class), method return type (some design pattern methods require a certain return type). In expert systems, there is no standard syntax for a rule. Rules can be created based on the experience of the knowledge engineer in a specific problem domain. However, rules should have machine processable representation. In addition, rules representation should be directive, readable and consistent. Rules and facts should be written in such a way that they can be matched by the inference engine.

5.3.1 Rules Template for Method Signatures of Design Patterns

A rules template has been created to reflect the required method signatures between pattern participant classes. We used readable and consistent rule syntax consistent with CLIPS rules syntax. In addition, we tried to represent the required method signatures of pattern participant classes. Table 5.1 shows the created rule syntax and its corresponding significance. The template has been created in such way that it complements the structure of each design pattern.

Rule Syntax	**Significance**
Class A *has method* m	Method m implemented inside class A
method m *returntype* Class A	Method m returns an object of type Class A
method m *Is_static* YES/NO	Method m is static or not
test (=(str-compare m1 m2)0)	To check whether m1 and m2 are the same methods (for overriding purposes)
test (neq m1 m2)	To check whether m1 and m2 are two different methods
method m1 *call_to* method m2	The implementation of method m1 involves a call to method m2

Table 5.1: The created rules template syntax and its significance

As Table 5.1 illustrates, all the required method signatures between pattern participant classes are represented by the rules template. Each rule has its condition and action parts. All the required method signatures are included in the condition parts. On the other hand, the action part indicates whether the instance is a true positive or a false positive. Some design patterns require methods call and methods overriding between participant classes, such as Adapter and Proxy. Other design patterns require methods return types, such as Prototype and Builder. The static signature (Is_static) has been used only once to filter the Singleton candidate instances. The Singleton design pattern requires a class to implement one static method with a Singleton return data type. The rules template is stored in the design patterns library to complement the structure representation of each design pattern. The MLDA rules/facts generator will generate the rules for the candidate instances based on the rules template. All template rules are consistent with the standard definition of design patterns presented by GoF.

Table 5.2 presents the created rules template for GoF design patterns. This table uses the rules syntax of Table 5.1. Each rule has title, condition and action parts. Each rule/action is enclosed in a parenthesis. Comments can be added when necessary after a semicolon. The rules template is consistent with CLIPS syntax. Hence, the generated rules can be loaded directly into CLIPS for processing.

Singleton_Rule	**Prototype_Rule**
1. (defrule Singleton_rule	1. (defrule Prototype_rule
2. IF	2. IF
3. (Singleton has method ?x)	3. (ConcretePrototype has method ?x)
4. (method ?x returntype Singleton)	4. (Prototype has method ?y)
5. (method ?x Is_static YES)	5. (test (=(str-compare ?x ?y)0))
6. THEN	6. (method ?x returntype Prototype)
7. (Singleton_instance is true positive)	7. THEN
8.) End Singleton Rule	8. (Prototype_instance is true positive)
	9.) End Prototype rule
AbstractFactory_Rule	**Factory_Rule**
1. (defrule AbstractFactory_rule	1. (defrule Factory_rule
2. IF	2. IF
3. (AbstractFactory has method ?x)	3. (Creator has method ?x)
4. (AbstractFactory has method ?y)	4. (ConcreteCreator has method ?y)
5. ((test(neq ?x ?y))	5. (test (= (str-compare ?x ?y) 0))
6. (ConcreteFactory has method ?x2)	6. (method ?x returntype Product)
7. (ConcreteFactory has method ?y2)	7. THEN
8. ((test(neq ?x2 ?y2))	8. (Factory_instance is true positive)
9. (test (= (str-compare ?x ?x2)0))	9.) End Factory rule
10. (test (= (str-compare ?y ?y2)0))	
11. THEN	
12. (AbstractFactory_instance is true positive)	
13.) End AbstractFactory rule	
Builder_rule	**Adapter_rule**
1. (defrule Builder_rule	1. (defrule Adapter_rule
2. IF	2. IF
3. (Builder has method ?x)	3. (Target has method ?x)
4. (ConcreteBuilder has method ?y)	4. (Adapter has method ?y)
5. (test (= (str-compare ?x ?y) 0))	5. (test (=(str-compare ?x ?y) 0))
6. (method ?x returntype Product)	6. (Adaptee has method ?z)
7. THEN	7. (method ?y call_to method ?z)
8. (Builder_instance is true positive)	8. THEN
9.) End Builder rule	9. (Adapter_instance is true positive)
	10.) End Adapter rule

Bridge_rule 1. (defrule Bridge_rule 2. IF 3. (ConcreteImplementor has method ?y) 4. (Implementor has method ?x) 5. (test (= (str-compare ?x ?y) 0)) 6. (Abstraction has method ?z) 7. (method ?z Call_to method ?x) 8. THEN 9. (Bridge_instance is true positive) 10.) End Bridge rule	**Composite_rule** 1. (defrule Composite_rule 2. (Leaf has method ?x) 3. (Component has method ?y) 4. (test (=(str-compare ?x ?y) 0)) 5. (Composite has method ?z) 6. (test (=(str-compare ?x ?z)0)) 7. THEN 8. (Composite_instance is true positive) 9.) End composite rule
Decorator_rule 1. (defrule Decorator_rule 2. (Component has method ?x) 3. (Decorator has method ?y) 4. (test (= (str-compare ?x ?y) 0)) 5. (ConcreteDecorator has method ?z) 6. (test (= (str-compare ?y ?z) 0)) 7. (method ?y call_to method ?x) 8. (method ?z call_to method ?y) 9. (ConcreteComponent has method ?q) 10. (test (= (str-compare ?z ?q) 0)) 11. THEN 12. (Decorator_instance is true positive) 13.) End Decorator rule	**Façade_rule** 1. (defrule Façade_rule 2. IF 3. (Façade has method ?x) 4. THEN 5. (Façade_instance is true positive) 6.) End of Facade rule
Command_rule 1. (defrule Command_Rule 2. IF 3. (Command has method ?x) 4. (ConcreteCommand has method ?y) 5. (test (= (str-compare ?x ?y)0)) 6. (Receiver has method ?z) 7. ((test(neq ?y ?z)) 8. (method ?y call_to method ?z) 9. THEN 10. (command_instance is true positive) 11.) End command rule	**Interpreter_rule** 1. (defrule Interpreter_rule 2. IF 3. (AbstractExpression has method ?x) 4. (NonTerminalExpression has method ?y) 5. (test (= (str-compare ?x ?y) 0)) 6. (method ?y call_to ?x) 7. THEN 8. (Interpreter_instance is true positive) 9.) End Interpreter rule
Iterator_rule 1. (defrule Iterator_rule 2. IF 3. (Iterator has method ?x) 4. (Aggregate has method ?y) 5. ((test(neq ?x ?y)) (ConcreteIterator has method ?z) 6. (test (= (str-compare ?x ?z) 0)) 7. (ConcreteAggregate has method ?q) 8. (test (= (str-compare ?y ?q) 0)) 9. (method ?y returntype Iterator) 10. THEN 11. (Iterator_instance is true positive) 12.) End Iterator instance	**Flyweight_rule** 1. (defrule Flyweight_rule 2. IF 3. (FlyweightFactory has method ?x) 4. (Flyweight has method ?y) 5. (method ?x returntype Flyweight) 6. ((test(neq ?x ?y)) 7. (UnsharedConcreteFlyweight has method ?z) 8. (test (= (str-compare ?y ?z) 0)) 9. (ConcreteFlyweight has method ?q) 10. (test (= (str-compare ?z ?q) 0)) 11. THEN 12. (flyweight_instance is true positive) 13.) End flyweight rule

Proxy_rule 1. (defrule Proxy_rule 2. IF 3. (Subject has method ?x) 4. (Proxy has method ?y) 5. (test (= (str-compare ?x ?y) 0)) 6. (RealSubject has method ?z) 7. (test (= (str-compare ?x ?z) 0)) 8. (method ?y call_to method ?z) 9. THEN 10. (Proxy_instance is true positive) 11.) End of proxy rule	**ChainOfResponsibility_rule** 1. (defrule CoR_rule 2. IF 3. (Handler has method ?x) 4. (ConcreteHandler has method ?y) 5. (test (=(str-compare ?x ?y)0)) 6. (method ?y call_to method ?x) 7. THEN 8. (CoR_instance is true positive) 9.) end of CoR rule
State/Strategy_rule 1. (defrule State/Strategy_Rule 2. IF 3. (Context has method ?x) 4. (State has method ?y) 5. ((test(neq ?x ?y)) 6. (ConcreteState has method ?z) 7. (test (=(str-compare ?y ?z)0)) 8. (method ?x call_to method ?y) 9. THEN 10. (State/Strategy_instance is true positive) 11.) End State/Strategy rule	**TemplateMethod_rule** 1. (defrule TemplateMethod_rule 2. IF 3. (ConcreteClass has method ?x) 4. (AbstractClass has method ?y) 5. (test (=(str-compare ?x ?y)0)) 6. THEN 7. (TemplateMethod_instance is true positive) 8.) End TemplateMethod rule
Visitor_rule 1. (defrule Visitor_rule 2. IF 3. (Element has method ?x) 4. (ConcreteElement has method ?y) 5. (test (=(str-compare ?x ?y)0)) 6. (Visitor has method ?z) 7. ((test(neq ?y ?z)) 8. (method ?y call_to method ?z) 9. THEN 10. (Visitor_instance is true positive) 11.) End of Visitor rule	**Mediator_rule** 1. (defrule Mediator_rule 2. IF 3. (Colleague has method ?x) 4. (method ?x returntype Mediator) 5. THEN 6. (Mediator_instance is true positive) 7.) End of mediator rule
Memento_rule 1. (defrule Memento_rule 2. IF 3. (Originator has method ?y) 4. (Caretaker has method ?x) 5. ((test(neq ?y ?x)) 6. (method ?x returntype Memento) 7. (ConcreteMemento has method ?z) 8. (test (=(str-compare ?y ?z)0)) 9. (method ?y call_to method ?z) 10. THEN 11. (Memento_instance is true positive) 12.) End memento rule	**Observer_rule** 1. (defrule Observer_rule 2. IF 3. (Subject has method ?x) 4. (Observer has method ?y) 5. ((test(neq ?x ?y)) 6. (ConcreteObserver has method ?z) 7. (test (=(str-compare ?y ?z)0)) 8. (method ?x call_to method ?y) 9. THEN 10. (Observer_instance is true positive) 11.) End observer rule

Table 5.2: Rules template for the method signatures of GoF design patterns

5.3.2 MLDA Rules/Facts Generator

The MLDA rules/facts generator – henceforth the R/F generator – is a simple Java program, implemented as part of the MLDA project, which generates a set of rules and facts to represent the required method signatures of candidate design instances and subject system respectively. Specifically, the R/F generator constructs a connection to the SQL tables, which hold the candidate design instances and the method signatures of the subject system, and uses Java Stream Writer, which is a Java library, to write the set of rules and facts. The outputs of the R/F generator are two files: rules.txt and facts.txt. These two files will be loaded later into CLIPS for processing.

5.3.2.1 Generating Rules

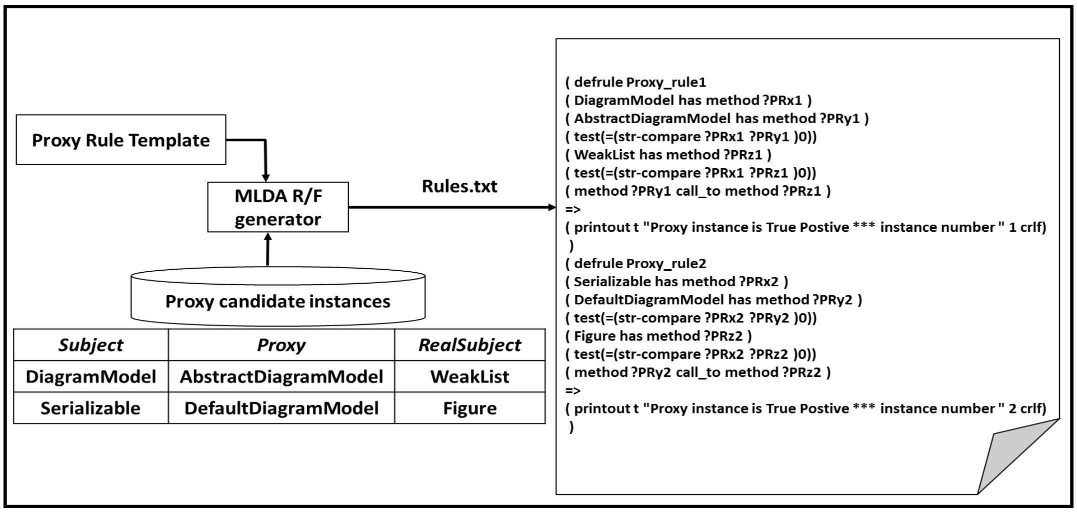
The candidate design pattern instances of each design pattern are stored in an SQL table. In order to generate the rules, the R/F generator constructs a connection to the SQL table and makes an access to each record. Based on the rules template, the R/F generator will generate a rule for each candidate design instance. Specifically, the R/F generator will fill each template entry by its corresponding role in the SQL table. Java "OutputStreamWriter" has been used to write the rules into a text file.

Figure 5.5 presents an example to illustrate how the R/F generator creates rules. The presented example shows the generated rules of the Proxy candidate instances. As Figure 5.5 illustrates, the R/F generator creates three rules to represent the required method signatures between Proxy participant classes (i.e. Subject, Proxy and RealSubject). Method IDs, titles and variables will be automatically incremented as new instances are inserted into the table. All the generated rules have a consistent syntax which can be loaded directly into CLIPS for processing. N candidate instances will be represented as N rules in the generated text file.

5.3.2.2 Generating Facts

The MLDA parser stores the retrieved method signatures from the subject system in an SQL table. The R/F generator represents each method signatures record as a set of facts. The generated facts are consistent with the rules syntax and conform to CLIPS syntax as well. The subject system is represented as a set of classes/interfaces where each record stores the methods implemented inside that class/interface. In addition, each record stores the method access modifier, return type, static status and method calls. Figure 5.6 shows an example of how the R/F generator generates facts to represent the method signatures of a subject system. Facts are generated based on the created facts template which is consistent with the rules template in such a way that the rule conditions can be matched to the generated facts.

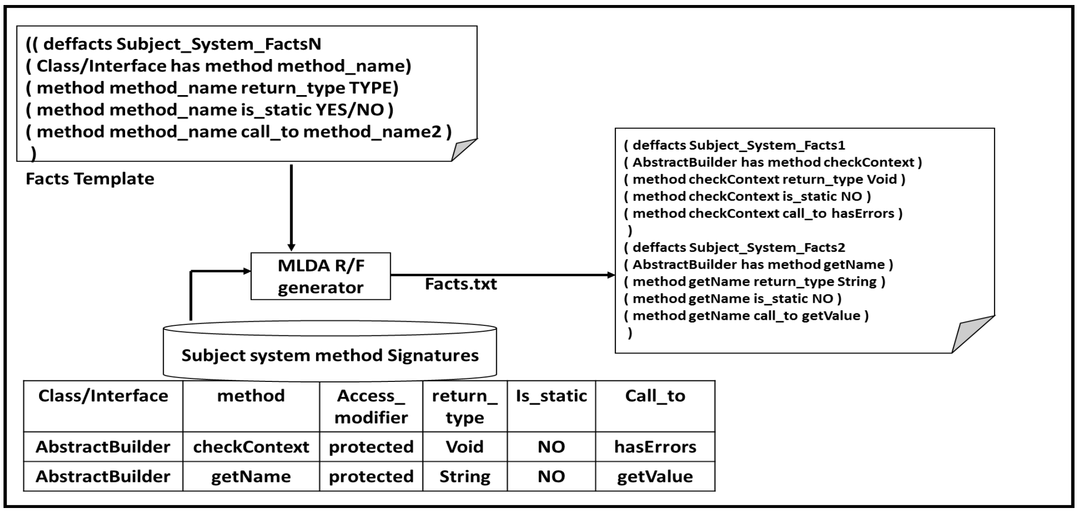
The R/F generator is customizable. This means that the syntax of the generated rules and facts can be changed by modifying the template.



Subject	*Proxy*	*RealSubject*
DiagramModel	AbstractDiagramModel	WeakList
Serializable	DefaultDiagramModel	Figure

```
( defrule Proxy_rule1
( DiagramModel has method ?PRx1 )
( AbstractDiagramModel has method ?PRy1 )
( test(=(str-compare ?PRx1 ?PRy1 )0))
( WeakList has method ?PRz1 )
( test(=(str-compare ?PRx1 ?PRz1 )0))
( method ?PRy1 call_to method ?PRz1 )
=>
( printout t "Proxy instance is True Postive *** instance number " 1 crlf)
 )
( defrule Proxy_rule2
( Serializable has method ?PRx2 )
( DefaultDiagramModel has method ?PRy2 )
( test(=(str-compare ?PRx2 ?PRy2 )0))
( Figure has method ?PRz2 )
( test(=(str-compare ?PRx2 ?PRz2 )0))
( method ?PRy2 call_to method ?PRz2 )
=>
( printout t "Proxy instance is True Postive *** instance number " 2 crlf)
 )
```

Figure 5.5: An example of the MLDA R/F generator for Proxy candidate instances



```
(( deffacts Subject_System_FactsN
( Class/Interface has method method_name)
( method method_name return_type TYPE)
( method method_name is_static YES/NO )
( method method_name call_to method_name2 )
 )
```

```
( deffacts Subject_System_Facts1
( AbstractBuilder has method checkContext )
( method checkContext return_type Void )
( method checkContext is_static NO )
( method checkContext call_to hasErrors )
 )
( deffacts Subject_System_Facts2
( AbstractBuilder has method getName )
( method getName return_type String )
( method getName is_static NO )
( method getName call_to getValue )
 )
```

Class/Interface	method	Access_ modifier	return_ type	Is_static	Call_to
AbstractBuilder	checkContext	protected	Void	NO	hasErrors
AbstractBuilder	getName	protected	String	NO	getValue

Figure 5.6: An example of the MLDA R/F generator for a subject system facts generation

5.3.3 Matching Rules and Facts

The generated facts and rules will be loaded to CLIPS for processing. Facts will be stored in the working memory while the rules will be stored in the knowledge base. The CLIPS inference engine uses forward chaining which relies on Rete algorithm to fire the rules. If rule conditions match a set of facts, the rule will be inserted into the agenda for execution. At the end of the cycle, all the matched rules will be in agenda. The inference engine will fire the rules based on their order in the knowledge base. The topmost rule will be executed first. However, the order of the rules in the knowledge base is not important. This is mainly due to the way that the rules template was created. More specifically, the action part of each rule does not assert new facts to the working memory. The action part only prints a message indicating that the instance is a true positive. Hence, the order of the rules will not affect the rules execution. One cycle is required to fire all rules. The “run” command would run the inference engine of CLIPS.

Rules that represent false positive instances should not be fired. These instances have a structure similar to that of GoF design patterns but they are not implementing the required method signatures of GoF design patterns. On the other hand, rules that represent true positive instances should be fired. However, some of the true positive instances are considered, by referring to the relevant literature, as true positive instances but are not implementing the required method signatures. Hence, these instances are partially implemented.

5.4 Summary

This chapter presented a rule-based approach to filter the false positive candidate instances detected by the MLDA’s structural search model.

The use of a rule-based approach aims to enhance the detection accuracy that relies on the principle of relationship matching. Specifically, the fundamental hypothesis that we want to explore is “whether representing the method signatures of the candidate design instances and subject system using a rule-based system is effective in improving the accuracy of design patterns recovery that relies on the relationship matching”.

The method signatures of GoF design patterns have a uniform structure and can be represented as an independent piece of knowledge, facilitating their representation as a set of rules.

Rules will represent the required method signatures between the participant classes of candidate instances. On the other hand, the method signatures of the subject system will be represented as a set of facts.

A rules template has been created as a base template to generate the rules and this template is consistent with the required method signatures of GoF design patterns. The MLDA rules/facts generator has been developed to generate the set of rules and facts, which will be directly loaded into CLIPS for processing. CLIPS uses forward chaining which relies on Rete algorithm to match the rules and facts.

Rules which represent the false positive instances should not be fired by the inference engine. This indicates that these instances have a structure similar to that of GoF design patterns but that they are not implementing the required method signatures such as method overriding and method calls. In contrast, rules which represent the true positive instances should be fired since they should implement the required method signatures. However, some of the true positive instances are partly implemented in the subject system and only they implement the required structure of design patterns.

Chapter Six

Applying a Rule-Based Approach

The candidate design pattern instances that have been recovered from eight subject systems after applying SSM will be filtered using a rule-based approach. The MLDA rules/facts generator will generate the required rules and facts to represent the method signatures of the candidate design instances and subject systems respectively. In addition, the CLIPS inference engine will process the generated facts and rules.

6.1 Introduction

SSM recovered the instances that have a structure similar to that of GoF. However, these instances resulted in many false positive instances. This is mainly due to the nature of object-oriented programs where many structures might be implemented in a way that is similar to that of GoF design patterns. Applying a rule-based approach aims to distinguish between GoF design structures and other structures in a subject system. This requires a set of facts, a set of rules and an inference engine to do the required processing.

The generated set of rules should reflect the required method signatures between pattern participant classes based on their standard definitions presented by GoF. On the other hand, the set of facts should reflect the existing method signatures inside the subject systems. However, applying a rule-based approach will cause an exact matching between the rule and fact sets. Exact matching indicates that an instance is said to be a true positive instance if its corresponding rule conditions can be met by a specific set of facts. The fact and rule templates have been constructed in such a way that they allow exact matching. The rules and facts template complements the generated library of design patterns. The Design patterns library reflects the structural features of design patterns, whereas the rules and facts template reflects their dynamic behavior. Hence, the required structural and behavioral features were considered.

This chapter presents the experimental results of applying a rule-based approach in attempts to enhance the accuracy of SSM.

6.2 Method Signatures Representation

The MLDA rules/facts generator (R/F generator) has been used to generate a set of rules and facts for each subject system and its recovered candidate instances. Once rule conditions are met by facts, a print message will be displayed to indicate that a candidate instance is a true positive. The number of generated facts and rules for all subject systems are presented in Table 6.1.

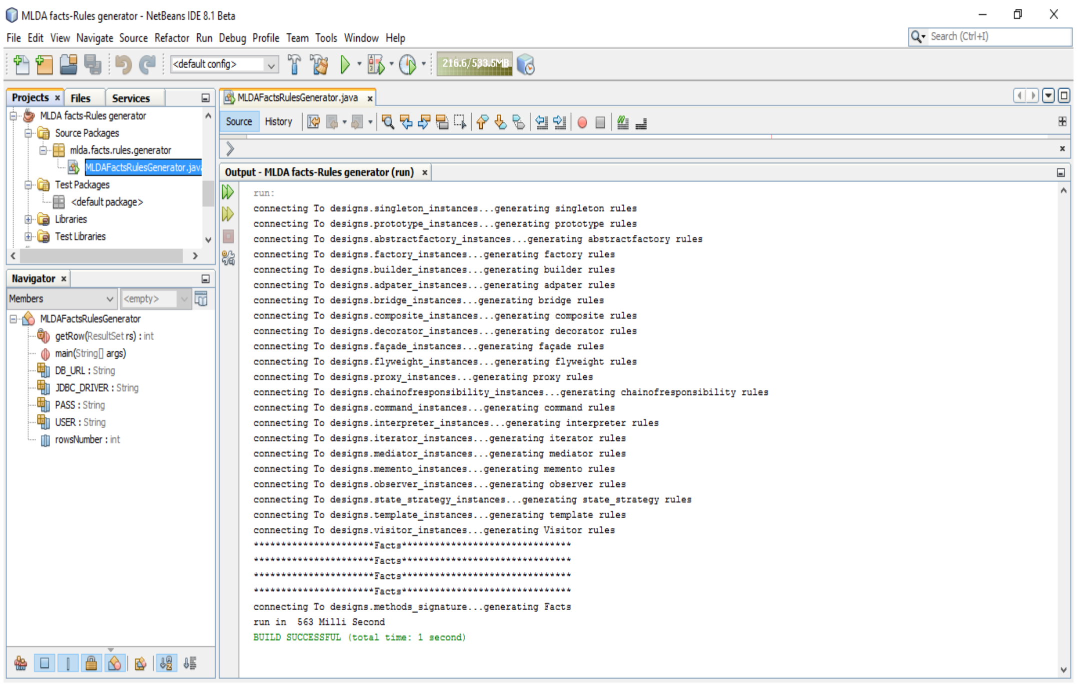
The number of generated facts depends on the number of implemented methods inside a subject system. As the number of implemented methods increases, the number of generated facts increases. On the other hand, the number of rules equals the number of candidate instances (i.e. each candidate instance is represented as one rule). The process of generating facts and rules is quite fast with the R/F generator spending only a few seconds on this. In addition, the MLDA parser spent around seven seconds recovering the methods from the subject systems. The R/F generator generates, in total, 43424 facts to represent the method signatures inside the subject systems and also generates, in total, 2994 rules to represent the required method signatures of the candidate design instances.

PMD has the largest number of facts and rules since it implements more methods than other subject systems and has too many structures similar to those of GoF design patterns. In contrast, JUnit and Lexi have the lowest number of facts and rules. In addition, the rules and facts generation time relies on the number of candidate instances and on the number of implemented methods inside a subject system.

Subject Systems	Number of facts	Number of rules	Generation time (seconds)	Methods parsing time (seconds)
JHotDraw	4222	195	0.5	1.40
JRefactory	9163	723	2.1	0.86
JUnit	1981	68	0.4	0.43
QuickUML	2397	185	0.4	0.54
Lexi	1664	216	0.5	0.17
MapperXML	6976	319	0.4	0.21
Nutch	6853	487	0.4	0.95
PMD	10168	801	0.7	2.2
Total	43424	2994	5.4	6.76

Table 6.1: The number of generated facts and rules

Figure 6.1 shows a screenshot of the process of running the MLDA R/F generator. The generator makes a connection onto each table that stores the candidate design instances, using an SQL connection command. In addition, the generator makes an access to a table that stores subject system method signatures. Then, based on the rules and facts templates, two text files (rules.txt and facts.text) will be automatically placed in the working directory of the MLDA project. These two files hold the method signatures representation for the candidate design instances and subject systems. The syntax of these two files is consistent in such a way that they can be loaded directly into CLIPS for processing.


MLDA facts-Rules generator - NetBeans IDE 8.1 Beta
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects Files Services
MLDA facts-Rules generator
Source Packages
mlda.facts.rules.generator
MLDAFactsRulesGenerator.java
Test Packages
<default package>
Libraries
Test Libraries
Navigator
Members
<empty>
MLDAFactsRulesGenerator
getRow(ResultSet rs) : int
main(String[] args)
DB_URL : String
JDBC_DRIVER : String
PASS : String
USER : String
rowsNumber : int
MLDAFactsRulesGenerator.java
Source History
Output - MLDA facts-Rules generator (run)

```
run:
connecting To designs.singleton_instances...generating singleton rules
connecting To designs.prototype_instances...generating prototype rules
connecting To designs.abstractfactory_instances...generating abstractfactory rules
connecting To designs.factory_instances...generating factory rules
connecting To designs.builder_instances...generating builder rules
connecting To designs.adpater_instances...generating adpater rules
connecting To designs.bridge_instances...generating bridge rules
connecting To designs.composite_instances...generating composite rules
connecting To designs.decorator_instances...generating decorator rules
connecting To designs.façade_instances...generating façade rules
connecting To designs.flyweight_instances...generating flyweight rules
connecting To designs.proxy_instances...generating proxy rules
connecting To designs.chainofresponsibility_instances...generating chainofresponsibility rules
connecting To designs.command_instances...generating command rules
connecting To designs.interpreter_instances...generating interpreter rules
connecting To designs.iterator_instances...generating iterator rules
connecting To designs.mediator_instances...generating mediator rules
connecting To designs.memento_instances...generating memento rules
connecting To designs.observer_instances...generating observer rules
connecting To designs.state_strategy_instances...generating state_strategy rules
connecting To designs.template_instances...generating template rules
connecting To designs.visitor_instances...generating Visitor rules
*********************Facts******************************
*********************Facts******************************
*********************Facts******************************
*********************Facts******************************
connecting To designs.methods_signature...generating Facts
run in  563 Milli Second
BUILD SUCCESSFUL (total time: 1 second)
```

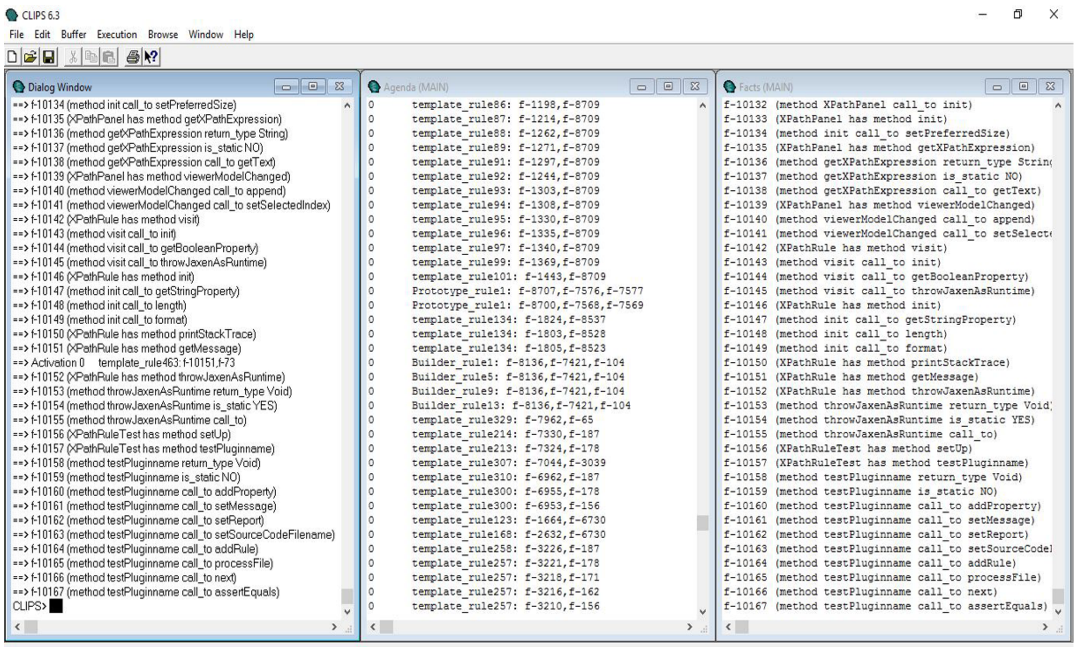
Figure 6.1: A screenshot for the running of the MLDA R/F generator

6.3 Rules and Facts Matching

The generated rules and facts will be loaded into CLIPS for processing. The inference engine of CLIPS will do the required “exact” matching between the set of rules and facts. Rules which represent the true positive instances should be fired. In contrast, rules which represent the false positive instances should not be fired.

6.3.1 CLIPS Processing

After loading the rules and facts files, the “reset” command is used to insert the facts into the working memory of CLIPS. The rules are automatically inserted into the knowledge base of CLIPS. All the matched rules will be inserted into the Agenda for execution. The “run” command will run the inference engine of CLIPS and all the rules in the Agenda will be executed. The output of the inference engine is a message which indicates that the instance is a true positive. Figure 6.2 shows a screenshot of CLIPS after loading the facts and rules files of PMD.



```
==> f-10134 (method init call_to setPreferredSize)
==> f-10135 (XPathPanel has method getXPathExpression)
==> f-10136 (method getXPathExpression return_type String)
==> f-10137 (method getXPathExpression is_static NO)
==> f-10138 (method getXPathExpression call_to getText)
==> f-10139 (XPathPanel has method viewerModelChanged)
==> f-10140 (method viewerModelChanged call_to append)
==> f-10141 (method viewerModelChanged call_to setSelectedIndex)
==> f-10142 (XPathRule has method visit)
==> f-10143 (method visit call_to init)
==> f-10144 (method visit call_to getBooleanProperty)
==> f-10145 (method visit call_to throwJaxenAsRuntime)
==> f-10146 (XPathRule has method init)
==> f-10147 (method init call_to getStringProperty)
==> f-10148 (method init call_to length)
==> f-10149 (method init call_to format)
==> f-10150 (XPathRule has method printStackTrace)
==> f-10151 (XPathRule has method getMessage)
==> Activation 0     template_rule463: f-10151,f-73
==> f-10152 (XPathRule has method throwJaxenAsRuntime)
==> f-10153 (method throwJaxenAsRuntime return_type Void)
==> f-10154 (method throwJaxenAsRuntime is_static YES)
==> f-10155 (method throwJaxenAsRuntime call_to)
==> f-10156 (XPathRuleTest has method setUp)
==> f-10157 (XPathRuleTest has method testPluginname)
==> f-10158 (method testPluginname return_type Void)
==> f-10159 (method testPluginname is_static NO)
==> f-10160 (method testPluginname call_to addProperty)
==> f-10161 (method testPluginname call_to setMessage)
==> f-10162 (method testPluginname call_to setReport)
==> f-10163 (method testPluginname call_to setSourceCodeFilename)
==> f-10164 (method testPluginname call_to addRule)
==> f-10165 (method testPluginname call_to processFile)
==> f-10166 (method testPluginname call_to next)
==> f-10167 (method testPluginname call_to assertEquals)
CLIPS>
```

```
0       template_rule86: f-1198,f-8709
0       template_rule87: f-1214,f-8709
0       template_rule88: f-1262,f-8709
0       template_rule89: f-1271,f-8709
0       template_rule91: f-1297,f-8709
0       template_rule92: f-1244,f-8709
0       template_rule93: f-1303,f-8709
0       template_rule94: f-1308,f-8709
0       template_rule95: f-1330,f-8709
0       template_rule96: f-1335,f-8709
0       template_rule97: f-1340,f-8709
0       template_rule99: f-1369,f-8709
0       template_rule101: f-1443,f-8709
0       Prototype_rule1: f-8707,f-7576,f-7577
0       Prototype_rule1: f-8700,f-7568,f-7569
0       template_rule134: f-1824,f-8537
0       template_rule134: f-1803,f-8528
0       template_rule134: f-1805,f-8523
0       Builder_rule1: f-8136,f-7421,f-104
0       Builder_rule5: f-8136,f-7421,f-104
0       Builder_rule9: f-8136,f-7421,f-104
0       Builder_rule13: f-8136,f-7421,f-104
0       template_rule329: f-7962,f-65
0       template_rule214: f-7330,f-187
0       template_rule213: f-7324,f-178
0       template_rule307: f-7044,f-3039
0       template_rule310: f-6962,f-187
0       template_rule300: f-6955,f-178
0       template_rule300: f-6953,f-156
0       template_rule123: f-1664,f-6730
0       template_rule168: f-2632,f-6730
0       template_rule258: f-3226,f-187
0       template_rule257: f-3221,f-178
0       template_rule257: f-3218,f-171
0       template_rule257: f-3216,f-162
0       template_rule257: f-3210,f-156
```

```
f-10132 (method XPathPanel call_to init)
f-10133 (XPathPanel has method init)
f-10134 (method init call_to setPreferredSize)
f-10135 (XPathPanel has method getXPathExpression)
f-10136 (method getXPathExpression return_type Strin
f-10137 (method getXPathExpression is_static NO)
f-10138 (method getXPathExpression call_to getText)
f-10139 (XPathPanel has method viewerModelChanged)
f-10140 (method viewerModelChanged call_to append)
f-10141 (method viewerModelChanged call_to setSelect
f-10142 (XPathRule has method visit)
f-10143 (method visit call_to init)
f-10144 (method visit call_to getBooleanProperty)
f-10145 (method visit call_to throwJaxenAsRuntime)
f-10146 (XPathRule has method init)
f-10147 (method init call_to getStringProperty)
f-10148 (method init call_to length)
f-10149 (method init call_to format)
f-10150 (XPathRule has method printStackTrace)
f-10151 (XPathRule has method getMessage)
f-10152 (XPathRule has method throwJaxenAsRuntime)
f-10153 (method throwJaxenAsRuntime return_type Void
f-10154 (method throwJaxenAsRuntime is_static YES)
f-10155 (method throwJaxenAsRuntime call_to)
f-10156 (XPathRuleTest has method setUp)
f-10157 (XPathRuleTest has method testPluginname)
f-10158 (method testPluginname return_type Void)
f-10159 (method testPluginname is_static NO)
f-10160 (method testPluginname call_to addProperty)
f-10161 (method testPluginname call_to setMessage)
f-10162 (method testPluginname call_to setReport)
f-10163 (method testPluginname call_to setSourceCode
f-10164 (method testPluginname call_to addRule)
f-10165 (method testPluginname call_to processFile)
f-10166 (method testPluginname call_to next)
f-10167 (method testPluginname call_to assertEquals)
```

Figure 6.2: A screenshot of CLIPS after loading the rules and facts of PMD

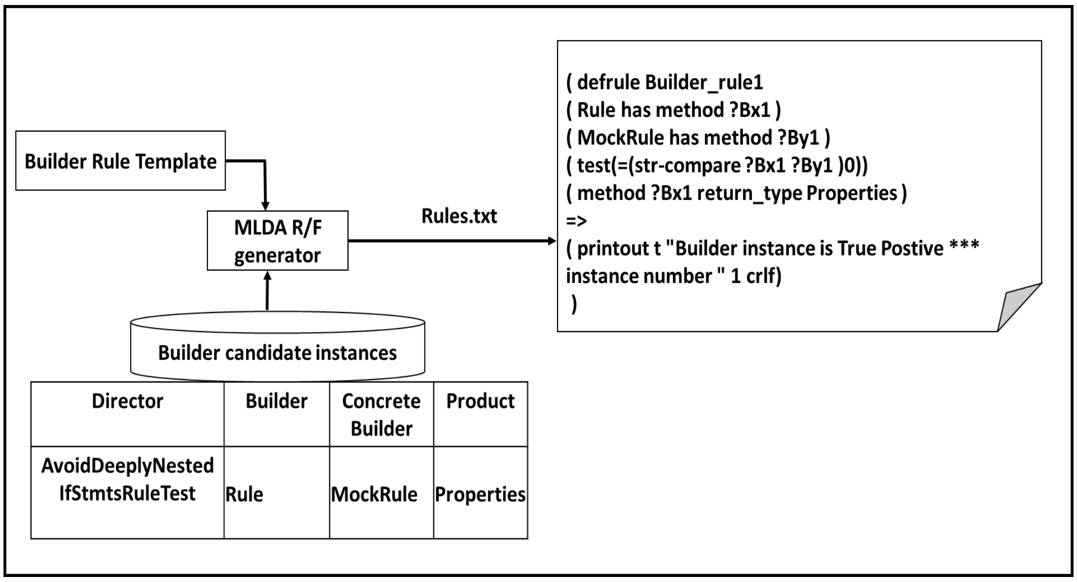
6.3.2 A PMD Example

To illustrate how CLIPS matches the set of facts and rules, the matching process for a Builder candidate instance, which is recovered from PMD, is presented. SSM recovered 16 Builder instances from PMD and these instances are true positive instances. Figure 6.3 shows a candidate Builder instance with its corresponding generated rule.

The generated rule reflects the required method signatures between Builder participant classes ("AvoidDeeplyNestedIfStmtsRuleTest", "Rule", "MockRule", and "Properties"). The rule suggests:

- Class "Rule" should implement a method;
- Class "MockRule" should implement the same method (overriding) that "Rule" implements;
- The return type of that method should be of type class "Properties".

Hence, all the required method signatures are reflected in the generated rule. However, reducing the number of rule conditions will enhance its opportunity to be matched with the set of facts. This may increase the number of false positive instances.



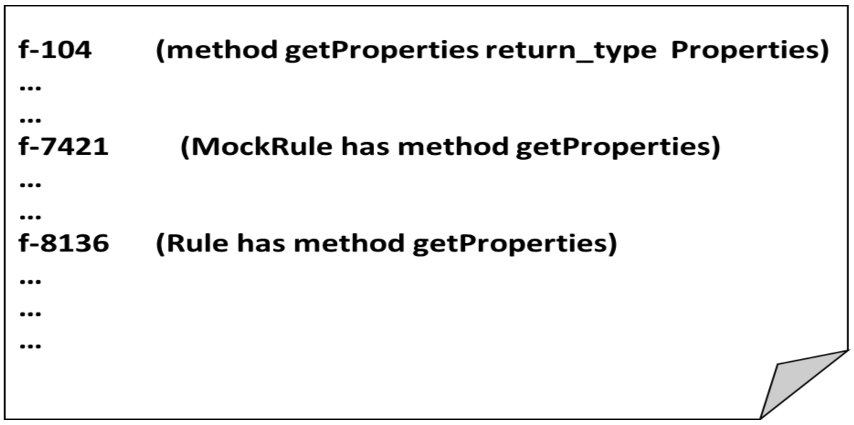
```
( defrule Builder_rule1
( Rule has method ?Bx1 )
( MockRule has method ?By1 )
( test(=(str-compare ?Bx1 ?By1 )0))
( method ?Bx1 return_type Properties )
=>
( printout t "Builder instance is True Postive ***
instance number " 1 crlf)
 )
```

Director	Builder	Concrete Builder	Product
AvoidDeeplyNested IfStmtsRuleTest	Rule	MockRule	Properties

Figure 6.3: A candidate Builder instance with its generated rule

The inference engine of CLIPS will match the rule conditions of the candidate Builder instance against the generated set of PMD facts. Figure 6.4 shows part of the generated facts of PMD after loading the facts file onto CLIPS. The presented facts are required to satisfy Builder rule conditions.

The inference engine of CLIPS will fire the Builder rule which represents the candidate instance of Figure 6.3. This indicates that this instance has a structure consistent with the structure of GoF and that it implements the required method signatures.



```
f-104      (method getProperties return_type  Properties)
...
...
f-7421       (MockRule has method getProperties)
...
...
f-8136     (Rule has method getProperties)
...
...
...
```

Figure 6.4: Part of the generated fact list of PMD

6.4 Accuracy Evaluation

The rule-based system has been applied to the recovered candidate instances after applying SSM. Tables 6.2 and 6.3 show the experimental results of recovering 23 design patterns from subject systems after applying the rule-based system.

6.4.1 Experiments and Results

The experimental results show that the method signatures of GoF design patterns provide an appropriate fingerprint for the detection of design pattern instances from object-oriented programs. Moreover, the third level of MLDA allows the detection of the template method design pattern. Specifically, the template method design pattern requires an inheritance relationship between two different classes. These two classes should implement the same method. This condition has been checked by the rule-based system and most of the false positive instances have been removed.

Furthermore, the experimental results show an enhancement in the detection accuracy for all subject systems. The recall rate did not affect this since it relies on the number of false negative instances. However, by referring to the relevant literature, some instances implement the required method signatures and these are not design patterns. MLDA recovered 61 such design instances. This is mainly due to the way that these instances were implemented. For example, one Command instance in JHotDraw has been implemented using two aggregation relationships and one inheritance relationship and it implements the required method signatures presented by GoF. This instance is not considered a true positive instance in the literature.

Subject Systems/ Design Patterns	JHotDraw							JRefactory							JUnit							QuickUML						
	CI	**DI**	**TP**	**FP**	**FN**	**P%**	**R%**	**CI**	**DI**	**TP**	**FP**	**FN**	**P%**	**R%**	**CI**	**DI**	**TP**	**FP**	**FN**	**P%**	**R%**	**CI**	**DI**	**TP**	**FP**	**FN**	**P%**	**R%**
Singleton	2	2	2	0	0	100	100	10	10	10	0	2	100	83	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Prototype	2	2	2	0	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Abstract Factory	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0
Factory Method	0	0	0	0	2	NA	0	101	81	79	2	8	98	91	2	2	2	0	0	100	100	12	12	12	0	6	100	67
Builder	0	0	0	0	0	NA	NA	0	0	0	0	2	NA	0	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0
Adapter	31	13	11	2	0	85	100	17	15	15	0	1	100	94	7	5	5	0	6	100	45	26	25	25	0	4	100	86
Bridge	7	5	4	1	0	80	100	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0
Composite	1	1	1	0	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	1	1	1	0	0	100	100
Decorator	1	1	1	0	2	100	33	1	1	1	0	0	100	100	1	1	1	0	0	100	100	2	2	2	0	1	100	67
Façade	1	1	1	0	0	100	100	2	2	2	0	NA	100	NA	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Flyweight	1	1	1	0	0	100	100	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Proxy	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	1	1	1	0	1	100	50
CoR	0	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Command	17	9	8	1	1	89	89	45	32	22	10	3	69	88	0	0	0	0	0	NA	NA	17	17	17	0	1	100	94
Interpreter	0	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Iterator	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	0	0	0	0	0	NA	NA
Mediator	2	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Memento	1	0	0	0	0	NA	NA	0	0	0	0	NA	NA	NA	0	0	0	0	0	NA	100	6	0	0	0	0	NA	NA
Observer	0	0	0	0	2	NA	0	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	1	1	1	0	0	100	100
State/Strategy	33	7	6	1	0	86	100	11	8	8	0	3	100	73	8	3	3	0	0	100	100	11	10	10	0	0	100	100
Visitor	1	1	1	0	1	100	50	1	1	1	0	1	100	50	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Template Method	95	5	4	1	0	80	100	535	21	4	17	0	19	100	50	2	1	1	0	50	100	105	13	3	10	0	23	100
Total/Average	195	48	42	6	8	94%	75%	723	171	142	29	20	83%	88%	68	13	12	1	9	92%	57%	185	85	75	10	16	88%	82%

Note:
CI: Candidate Instances after applying Structural Search Model (level two) **DI**: Detected Instances after applying the method signatures matching level (level three)
P: Precision% **R**: Recall% **NA**: Not Applicable
TP: True Positives **FP**: False Positives **FN**: False Negatives

Table 6.2: The experimental results of recovering 23 GoF design patterns after applying the rule-based system-part 1

Subject Systems/ Design Patterns	Lexi							MapperXML							Nutch							PMD						
	CI	DI	TP	FP	FN	P%	R%	CI	DI	TP	FP	FN	P%	R%	CI	DI	TP	FP	FN	P%	R%	CI	DI	TP	FP	FN	P%	R%
Singleton	2	2	2	0	0	100	100	3	3	3	0	0	100	100	2	2	2	0	0	100	100	2	2	2	0	2	100	50
Prototype	1	1	1	0	0	100	100	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Abstract Factory	0	0	0	0	0	NA	NA	0	0	0	0	2	NA	0	0	0	0	0	0	NA	NA	0	0	0	0	2	NA	0
Factory Method	26	0	0	0	0	NA	NA	0	0	0	0	22	NA	0	8	5	5	0	0	100	100	7	7	7	0	25	100	22
Builder	13	0	0	0	0	NA	NA	2	1	1	0	0	100	100	0	0	0	0	0	NA	NA	16	16	16	0	0	100	100
Adapter	44	9	9	0	21	100	30	17	5	5	0	36	100	12	63	44	44	0	0	100	100	53	42	42	0	45	100	48
Bridge	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Composite	3	3	3	0	0	100	100	0	0	0	0	1	NA	0%	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Decorator	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	4	NA	0	0	0	0	0	0	NA	NA
Façade	1	1	1	0	0	NA	NA	1	1	1	0	0	100	100	1	1	1	0	0	100	100	1	1	1	0	0	100	100
Flyweight	1	1	1	0	0	100	100	1	1	1	0	0	100	100	1	1	1	0	0	100	100	0	0	0	0	0	NA	NA
Proxy	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	4	4	4	0	0	100	100	3	3	3	0	1	100	75
CoR	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	1	1	1	0	0	100	100
Command	15	6	6	0	0	100	100	52	26	26	0	0	100	100	23	23	23	0	2	100	92	3	3	3	0	0	100	100
Interpreter	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Iterator	36	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0	0	0	0	0	0	NA	NA
Mediator	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA
Memento	10	10	10	0	0	NA	NA	12	12	12	0	0	100	100	1	1	1	0	0	100	100	1	1	1	0	0	100	100
Observer	0	0	0	0	0	NA	NA	0	0	0	0	3	NA	0	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0
State/Strategy	0	0	0	0	1	NA	0	11	11	11	0	0	100	100	124	116	113	3	0	97	100	248	21	21	0	0	100	100
Visitor	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	0	NA	NA	0	0	0	0	1	NA	0
Template Method	64	0	0	0	0	NA	NA	220	56	56	0	0	100	100	260	7	7	0	0	100	100	464	120	108	12	0	90	100
Total/Average	216	33	33	0	22	100%	50%	319	116	116	0	64	100%	64%	487	204	201	3	7	99%	97%	801	219	207	12	77	95%	73%

Note:
CI: Candidate Instances after applying Structural Search Model (level two) **DI**: Detected Instances after applying the method signatures matching level (level three)
P: Precision% **R**: Recall% **NA**: Not Applicable
TP: True Positives **FP**: False Positives **FN**: False Negatives

Table 6.3: The experimental results of recovering 23 GoF design patterns after applying the rule-based system-part 2

6.4.2 Average Accuracy

The average precision achieved by MLDA after applying the rule-based system for the recovery of 23 design patterns from all subject systems is 93%. The rule-based approach removes most of the false positive instances detected by SSM. Table 6.4 shows a summary for the achieved detection accuracy after applying the hybrid structural rule-based approach. As the results illustrate, representing the method signatures of the candidate design instances and the subject system as a set of rules and facts enhances the detection accuracy of design patterns, which relies on the principle of relationship matching.

	After applying SSM								After applying the rule-based approach				
	All GoF design patterns				All GoF design patterns Except Template method design pattern				All GoF design patterns				
Subject Systems	CI	TP	FP	FN	CI	TP	FP	FN	CI	DI	TP	FP	FN
JHotDraw	195	42	153	8	100	38	62	8	195	48	42	6	8
JRefactory	723	142	581	20	188	138	50	20	723	171	142	29	20
JUnit	68	12	56	9	18	11	7	9	68	13	12	1	9
QuickUML	185	75	110	16	80	72	8	16	185	85	75	10	16
Lexi	216	22	194	22	152	22	130	22	216	33	33	0	22
MapperXML	319	116	203	64	99	60	39	64	319	116	116	0	64
Nutch	487	201	286	7	227	194	33	7	487	204	201	3	7
PMD	801	207	594	77	337	99	238	77	801	219	207	12	77
Total	2994	817	2177	223	1201	634	567	223	2994	889	828	61	223
Average precision	27%				53%				93%				
Average recall	79%				74%				79%				
Average F-measure	41%				62%				85%				

Note:
CI: Candidate Instances after applying Structural Search Model (level two)
DI: Detected Instances after applying a rule-based appraoch (level three)
TP: True Positives **FP**: False Positives **FN**: False Negatives

Table 6.4: Summary of the average accuracy achieved after applying SSM and the rule-based approach

6.5 Results Comparison

The accuracy of MLDA has been compared to four approaches, as presented in Tables 6.5 and 6.6. The selection of these approaches was made based on their results which were detailed enough to compare and were applied to the same subject systems (JHotDraw version 5.1 and JUnit version 3.7). However, the comparison among design pattern detection approaches is challenging. This is due to the fact that there is no standard benchmark to validate the results of each approach. In fact, each approach has its limitations, patterns representation, subject systems and validation method. Tables 6.5 and 6.6 show the results of the design patterns recovery of MLDA, Sempatrec [44], DeMIMA [41], Sub-patterns [33] and SSA [31] for JHotDraw and JUnit respectively. As illustrated, MLDA achieves reasonable detection accuracy in terms of precision for the detection of JHotDraw and JUnit instances.

The positiveness of MLDA relies on its ability to build the structure of each design pattern, record all the object interactions and match the method signatures. Hence, increasing the number of true positive instances. More specifically, MLDA recovers the five key relationships that may occur between classes and interfaces inside an object-oriented program, tries all the possible combinations between the recovered classes and interfaces and matches these structures to that of GoF. In addition, MLDA matches the method signatures of the candidate design instances to that of the subject system. Representing the method signatures using a rule-based approach provides a fingerprint for design patterns inside the source code. Both the structure and the method signatures are required to detect accurately the instances of GoF design patterns. The rule-based system enhances the detection accuracy of design patterns, which relies on the relationship matching.

However, MLDA missed the instances partly implemented in the source code, since SSM relies on the standard definition of GoF. On the other hand, the lack of dynamic and runtime information explains the existence of false positives. It must be noted that we only compare the results that DeMIMA, SSA, Sempatrec, and Sub-patterns revealed.

DPs	SS	MLDA			DeMIMA			SSA			Sempatrec			Sub-patterns		
		P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
AD	JD	85	100	92	4	100	8	44	100	61	45			100	NA	NA
DE	JD	100	33	50	8	100	15	33	33	33	50	33	40	100	NA	NA
CO	JD	89	89	89	33	100	50	100	100	100	100	100	100	100	NA	NA
FM	JD	NA	0	NA	2	100	4	100	67	80	100	100	100	100	NA	NA
SI	JD	100	100	100	100	100	100	100	100	100	100	100	100	100	NA	NA
OB	JD	NA	0	NA	25	100	40	50	40	44	50	40	44	100	NA	NA
TM	JD	80	100	89	7	100	13	20	100	33	50	100	67	100	NA	NA
VI	JD	100	50	67		100		100	100	100				100	NA	NA
Average %		**92**	**63**	**81**	**34**	**100**	**47**	**74**	**88**	**75**	**83**	**87**	**85**	**100**	**NA**	**NA**

Note:
AD: Adapter DE: Decorator CO: Command FM: Factory Method SI: Singleton OB: Observer TM: Template Method
VI: Visitor SS: Subject Systems JD: JHotDraw JU: JUnit P:Prescion R: Recall F: F-measure
Blank: Not Revealed NA: Not Applicable since the number of detected instances is zero or there is no reference to validate the instances

Table 6.5: Comparison of the results of MLDA and that of other approaches for JHotDraw

DPs	SS	MLDA			DeMIMA			SSA			Sempatrec			Sub-patterns		
		P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
AD	JU	100	45	62	0			17	100	29	100	100	100	100	100	100
DE	JU	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
CO	JU	NA	NA	NA	100	100	100	100	100	100	100	100	100	100	100	100
FM	JU	100	100	100		100								100	100	100
SI	JU	NA	NA	NA										100	100	100
OB	JU	NA	0	NA	25	100	40	100	100	100	100	100	100	100	50	67
TM	JU	50	100	67	0			100	100	100	100	100	100	100	100	100
VI	JU	NA	NA	NA		100									100	100
Average %		**88**	**69**	**82**	**45**	**100**	**80**	**83**	**100**	**86**	**100**	**100**	**100**	**100**	**94**	**96**

Note:
AD: Adapter DE: Decorator CO: Command FM: Factory Method SI: Singleton OB: Observer TM: Template Method
VI: Visitor SS: Subject Systems JD: JHotDraw JU: JUnit P:Prescion R: Recall F: F-measure
Blank: Not Revealed NA: Not Applicable since the number of detected instances is zero or there is no reference to validate the instances

Table 6.6: Comparison of the results of MLDA and that of other approaches for JUnit

Furthermore, the accuracy of MLDA has been compared to the sub-patterns approach for 23 design patterns since it is the only approach that recovers all GoF design patterns. Table 6.7 presents that comparison. The average accuracy of MLDA for JHotDraw and JUnit was 93% and 90% respectively. On the other hand, the average accuracy of the sub-patterns approach for JHotDraw and JUnit was 100% and 84% respectively.

	MLDA						Sub-patterns					
	JHotDraw			JUnit			JHotDraw			JUnit		
Design Patterns	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
Singleton	100	100	100	NA	NA	NA	100	NA	NA	100	100	100
Prototype	100	100	100	NA	NA	NA	100	NA	NA	NA	NA	NA
Abstract Factory	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Factory Method	NA	0	NA	100	100	100	100	NA	NA	100	100	100
Builder	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Adapter	85	100	92	100	45	62	100	NA	NA	100	100	100
Bridge	80	100	89	NA	NA	NA	100	NA	NA	NA	NA	NA
Composite	100	100	100	NA	0	NA	100	NA	NA	NA	NA	NA
Decorator	100	33	50	100	100	100	100	NA	NA	100	100	100
Façade	100	100	100	NA	NA	NA	100	NA	NA	NA	NA	NA
Flyweight	100	100	100	NA	NA	NA	100	NA	NA	NA	NA	NA
Proxy	NA	NA	NA	NA	NA	NA	100	NA	NA	NA	NA	NA
Chain of Responsibility	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Command	89	89	89	NA	NA	NA	100	NA	NA	NA	NA	NA
Interpreter	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Iterator	NA	NA	NA	NA	0	NA	NA	NA	NA	NA	NA	NA
Mediator	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Memento	NA	NA	NA	NA	100	NA	NA	NA	NA	NA	NA	NA
Observer	NA	0	NA	NA	0	NA	100	NA	NA	50	50	50
State_Strategy	86	100	93	100	100	100	100	NA	NA	40	NA	NA
Visitor	100	50	67	NA	NA	NA	100	NA	NA	NA	NA	NA
Template method	80	100	89	50	100	67	100	NA	NA	100	100	100
Average%	**93**	**77**	**89**	**90**	**61**	**86**	**100**	**NA**	**NA**	**84**	**92**	**92**
Note: P:Prescion R: Recall F: F-measure NA: Not Applicable since the number of detected instances is zero or there is no reference to validate the instances												

Table 6.7: Comparison of the results of MLDA and that of the sub-patterns approach for 23 design patterns

6.6 Summary

This chapter presented the experimental results of applying a rule-based system aimed at enhancing detection accuracy that relies on the relationships matching. The rule-based system provides a consistent format to represent the method signatures of the candidate design instances and the subject system. More specifically, the CLIPS inference engine has been used to match the set of rules and facts. Once all the rules' conditions are met by subject system facts, a message indicates that this instance is a true positive instance and will be printed.

As the experimental results illustrate, the rule-based system is able to reduce the number of false positive instances. These instances have a structure similar to that of GoF, but they fail to implement the required method signatures.

Chapter Seven
Design Patterns Impact

The impact of GoF design patterns on software quality attributes provides a support for decision-making during software design and refactoring. Researchers attempted to investigate the impact of applying design patterns on software quality by empirical methods such as case studies, surveys and experiments. Unfortunately, safe conclusions could not be drawn since the reported results are controversial.

The impact of design patterns on software quality is governed by a number of factors such as pattern variants, developer experience and quality that must be achieved by the pattern. In addition, the application of one pattern enhances certain quality attributes and simultaneously decreases others. The implementation of design patterns can vary across studies and these variants could be responsible for any differences observed in the reported results of the effects of design patterns on quality attributes. This chapter aims to assess quantitatively the impact of software design patterns on software maintainability and understandability using software metrics and design-pattern occurrences. Our use of these quality attributes relies on their definitions and features as they are presented by the ISO/IEC 9126 quality model [14]. ISO/IEC 9126 enjoys the benefits of being an international standard agreed by the software engineering community.

7.1 Introduction

Design patterns are the focus of many works studying their relevance, visualization and identification, with the hypothesis that their use improves quality. Gamma *et al.* claim in the preface of their book [11]: “You will have insight that makes your own designs more flexible, modular, reusable and understandable”.

Gamma *et al.* [11] describe through discussions how design patterns support adaptability and are expected to promote software evolution; they easy maintenance tasks by explicitly identifying class roles and by localizing where extensions and change should occur. However, the authors do not demonstrate the benefits to real software development projects. One benefit, for example, is that design patterns promote adaptability by supporting modifications through specialization. Developers can adapt a system built using these patterns by creating new concrete classes with desired functionality rather than by direct modifications to existing classes.

However, design patterns usually lead to an increased number of software artifacts, such as classes, associations and delegations, which increase the static complexity of a software system. Moreover, when the additional associations are instantiated at run-time, they result in additional links between objects which increase the dynamic complexity of a software system. A relevant benefit of design patterns is resilience to changes, avoiding that new requirement and, in general, any kind of system evolution causes major re-design. In addition, quality aspect advantages of design patterns include decoupling a request from specific operations (Chain of responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), making a system independent of algorithmic solutions (Iterator, Strategy and Visitor) and avoiding modifying implementations (Adapter, Decorator and Visitor). Since design patterns influence the system structure and their implementations are influenced by it, pattern implementations are often tailored to the instance of use. This makes it hard to distinguish between the pattern, the concrete instance and the object model involved. In fact, the application of patterns involves the introduction of new classes/interfaces and the moving of methods and requires additional code.

This chapter aims to address whether the classes playing roles in design patterns have better software metrics than do other classes in the system. Higher values of certain software metrics indicate good quality (e.g. cohesion). In contrast, higher values of other software metrics indicate bad quality (e.g. coupling between objects). This will provide an indication of how the implementation of design pattern instances affects the quality of a subject system. The main motivation behind the use of software metrics to quantify the subject system is their ability to provide a static and stable representation of the subject system. Moreover, since MLDA uses a class-level representation of the subject system and the recovered design instances have been validated based on the all publicly available results in the literature, *a list of classes playing roles in design patterns was generated for all subject systems*. This will facilitate the use of class-level metrics to quantify each class in the subject system. Hence, software metrics can be calculated for classes playing roles in design patterns and can be compared with classes that don't play roles in design patterns. Previous research studies which correlate class-level metrics to software quality might be used in attempts to address the impact of design patterns on software quality attributes. More specifically, we will examine the impact of design patterns on understandability and maintainability (whether they are positive or negative or whether they have no impact). The selection of these attributes was made since they are the most commonly investigated quality attributes, concerning design patterns' impact, in the available literature.

The remaining of this chapter is organized as follows: Section Two outlines a methodology to address the impact of design patterns, using software metrics and design pattern occurrences, on software maintainability and understandability. The experiments and results are presented in Section Three. Finally, threats to validity and conclusions are summarized in Sections Four and Five respectively.

7.2 Methodology

There is no formal theory that links design patterns to software quality concepts. However, it has been claimed that the use of design patterns provides several advantages, such as increased reusability and improved maintainability and comprehensibility of existing systems.

The methodology presented relies on software metrics, which are useful measurements for characterizing software systems, in an attempt to assess the impact of design patterns on software maintainability and understandability. Based on their scope, software metrics can be divided into two categories [97]:

- Project metrics: these metrics deal with the dynamics of a project and with what is needed to reach a certain point in the development life cycle. Project metrics provide a higher level of abstraction of the system. They can be used for estimation purposes such as estimating the number of required staff.
- Design metrics: these metrics deal with assessing the size, quality and complexity of software systems. Design metrics use the source code as input to quantify the system design and are more locally and specifically focused.

Software design metrics for all system classes, at the class level, will be calculated to investigate whether a safe conclusion can be drawn regarding the impact of design patterns on software maintainability and understandability. To calculate the required metrics, a Java metrics tool named JHawk has been used. We used the latest version of JHawk, v6.1.3, under the academic license granted from virtual machinery [98].

7.2.1 JHawk Tool

JHawk is a Java metrics tool that can evaluate the static behavior of the Java source code [98]. It collects metrics at four different levels. The lowest level is the method level, followed by the class level, the package level and the system level. More specifically, JHawk uses the source code of the subject system as input and calculates its metrics based on numerous aspects of the code, such as volume, complexity, relationships between class and packages and relationships within classes and packages. JHawk metrics could be used to capture poor design, poor coding practice, fault prediction, simple quantity measurements and overall code quality. JHawk classifies its metrics into three categories: quantitative metrics which measure the quantity of the source code; complexity metrics which measure how complex the code is in terms of its functionality and readability complexity; and structural metrics which measure the contribution of individual code artifacts to the quality of the code.

The selection of JHawk to calculate the required metrics was made since it has been used by a significant number of academic studies on Java metrics. In addition, JHawk provides around 103 different Java source code metrics at different levels (system, package, class and method). It has unrivaled accuracy and acceptable ranges can be set for particular metrics and for packages, methods and classes. Furthermore, JHawk allows new metrics to be created and added to its metric set. Reports can be output in HTML, CSV and XML formats.

7.2.2 A Metrics-Based Approach

Figure 7.1 presents a metrics-based approach to assess the impact of design pattern instances on software quality attributes. As mentioned above, all metrics will be calculated using JHawk at the class level.

The steps of the metrics-based approach can be summarized as follows:

1. Recover design pattern instances from each subject system.
2. For each design instance, determine its participant classes.
3. Classify system classes into two groups: classes that play roles in design patterns (henceforth pattern classes) and classes that do not (henceforth non-pattern classes).
4. Calculate size, coupling and inheritance metrics for all system's classes.
5. For each class in the system, calculate the percentage of participation in the total metric value.
6. Calculate the percentage of participation in the total value of each metric for both groups: pattern classes and non-pattern classes.

7. Correlate software metrics to quality attributes using previous research studies.

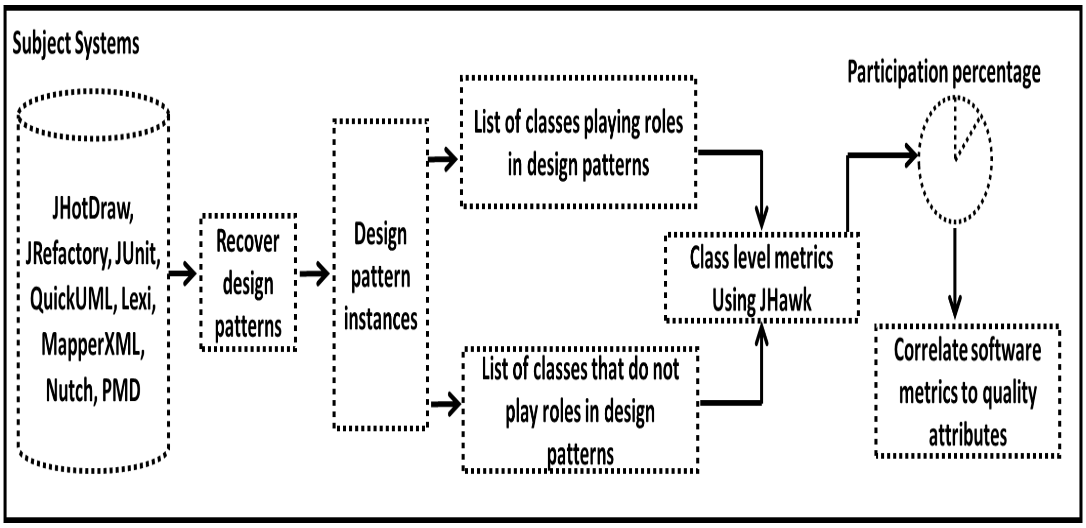

Subject Systems
JHotDraw, JRefactory, JUnit, QuickUML, Lexi, MapperXML, Nutch, PMD
Recover design patterns
Design pattern instances
List of classes playing roles in design patterns
List of classes that do not play roles in design patterns
Class level metrics Using JHawk
Participation percentage
Correlate software metrics to quality attributes

Figure 7.1: A metrics-based approach to assess the impact of design patterns

Based on the validated set of design pattern instances recovered using MLDA, all classes participating in design patterns were determined in eight subject systems. Furthermore, the following metrics will be calculated [99], [100]:

- *Number of methods (NOM)*: counts the total number of methods implemented inside a given class.
- *Lack of Cohesion of Methods (LCOM)*: counts the sets of methods in a given class that are not related through the sharing of some of the class's fields.
- *Total Response For Class (RFC)*: measures the number of different methods that can be executed when an object of a given class receives a message (when a method is invoked for that object).
- *Coupling Between Objects (CBO)*: counts the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types and exceptions.

- *Total Lines of Code* (LOC): counts the total lines of code inside a given class (excluding comments and blanks).
- *Fan-IN (F-IN)*: counts the number of classes calling methods implemented in a given class.
- *Depth of Inheritance Tree (DIT)*: counts the maximum inheritance path from a given class to the root class in the inheritance hierarchy.
- *Cohesion (COH)*: counts the degree to which the elements inside a given class belong together.
- *Fan-out (FOUT)*: counts the number of classes called by methods implemented in a given class (i.e. the number of classes that a given class uses and not the classes it is used by).
- *Number of Children (NOC)*: counts the number of direct sub-classes of a given class.

The selection of these metrics was made since they have a key role in characterizing the quality of software systems. More specifically, these metrics reflect the main aspects of any object-oriented program, i.e. size, coupling and inheritance [97], [99]. The input of the metrics-based approach is the source code of the subject system, whereas the output is the percentage of participation for both groups in the total metric value. The participation percentage of each individual class is calculated based on the total system metric value. For example, if the total system coupling was 1396 and a class has a coupling value of 65, then its participation percentage in the total system coupling can be calculated as (65 / 1396) × 100% = 4.66%. Hence, the participation percentage for all classes in the system can be calculated.

7.2.3 Correlation of Software Metrics to Quality Attributes

The external behavior of software systems can be recognized based on its internal metrics. Several studies were presented in the literature to correlate the impact of calculated software metrics with quality attributes. This correlation was made based on certain statistical analysis and experiments.

The impact can be positive or negative, or there can be no impact at all. We are trying to address the impact of design patterns on software understandability and maintainability since they are the most commonly investigated quality attributes. Table 7.1 illustrates the correlated impact of NOM, LOC, RFC, CBO, LCOM, COH, F-IN, FOUT, DIT and NOC on software understandability and maintainability. This correlation was made based on the correlation presented by [101], [102], [103], [104] and [105]. These studies were selected since they have significant correlation levels. However, to the best of our knowledge, there are no studies that contradict the reported impact presented by the selected studies.

	Size		**Coupling**						**Inheritance**	
Metrics\ Quality attributes	**NOM**	**LOC**	**RFC**	**CBO**	**LCOM**	**COH**	**F-IN**	**FOUT**	**DIT**	**NOC**
Understandability	▼	▼	▼	▼	▼	▲	▼	◄►	▼	▼
Maintainability	▼	▼	▼	▼	▼	▲	▼	◄►	▼	▼

▲: Positive impact ▼: Negative impact ◄►: No impact
NOM: Number of Methods **LOC**: Lines Of Codes
RFC: Response For Class **CBO**: Coupling Between Objects
LCOM: Lack of Cohesion in Methods **COH**: Cohesion
F-IN: Fan-IN **FOUT**: Fan-out
DIT: Depth of Inheritance Tree **NOC**: Number of Children

Table 7.1: The correlation between software metrics and software understandability and maintainability

The positive impact (▲) of a metric indicates that high values of that metric are desirable. On the other hand, a metric's negative impact (▼) indicates that high values of that metric are not desirable. Table 7.1 demonstrates that most of the selected metrics have a negative impact on software maintainability and understandability (i.e. high values of these metrics is a sign of bad quality). The cohesion metric has been reported to have a positive impact on understandability and maintainability efforts, whereas FOUT has no impact at all.

NOM inside a given class is a predictor of how much time and effort is required to understand and maintain the class. Classes with large numbers of methods are likely to be more application specific and, hence, more maintainability and understandability effort is required. In addition, the greater the number of methods in a class, the greater the potential impact on children, since children will inherit all the methods defined in the class.

LOC physically measures the total number of Java statements, excluding comments and blanks, inside a given class. LOC is an estimation metric which gives an indication of the amount of effort required to develop a software. However, LOC can be hardly used to estimate the developers' productivity and/or the functionality of a software, since skilled developers may be able to develop the same functionality with less code. In general, as the number of lines of codes increases, the maintainability and understandability efforts also increase.

RFC is a set of methods that can potentially be executed in response to a message received by an object of a class. It is calculated by adding the number of methods in the class, not including inherited methods, plus the number of distinct method calls made by the methods in the class (each method call is counted only once even if it is called from different methods). This can be done by inspecting method calls within the class's method bodies. If a large number of methods can be invoked in response to a message, then the maintainability and understandability efforts become more complicated (i.e. the larger number of methods that can be invoked from a class, the greater the complexity of the class).

CBO provides an indication of the strength of interconnections between system classes. Higher values of this metric indicate that more maintainability and understandability efforts are required. The tightly coupled system means that its classes are dependent on each other. In general, classes are tightly coupled if they use shared variables or if they exchange control information. A particular class in a tightly coupled system might be harder to reuse and test since dependent classes must be included. In contrast, loosely coupled systems mean that classes are independent and can function completely without the presence of the others. However, it is difficult to find a system with classes that are completely independent. Low coupling is a sign of a good design and a well-structured system.

LCOM measures the correlation between the methods and the local instance variables of a class. It is viewed as a measure of how well the methods of the class co-operate to achieve the aims of the class. LCOM is calculated as the ratio of methods in a class that do not access a specific data field, averaged over all data fields in the class. More specifically, LCOM is a count of the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero. If none of the methods of a class display any instance behavior, i.e. they do not use any instance variables, then they have no similarity and the LCOM value for the class will be zero. Lower values of LCOM are desirable (the class is more cohesive). Higher values of LCOM indicate that the system needs greater maintainability and understandability efforts and that the classes should be split into two or more sub-classes. LCOM can be in the range of 0 to 2 with values over 1 viewed as being suggestive of poor design [100].

COH is used to indicate the degree to which a class has a single and well-focused purpose. It refers to the degree to which the elements inside a class belong together. A cohesive class performs one function. Lack of cohesion means that a class performs more than one function. High cohesion is desirable since it promotes a lesser maintainability and understandability effort. Cohesion is increased if the class methods carry out a small number of related activities. Cohesion is often contrasted with coupling (i.e. high cohesion often correlates with loose coupling and vice versa) [105].

F-IN measures the number of other classes that reference a given class. In contrast, FOUT measures the number of other classes referenced by a given class. A high F-IN indicates a heavily used class. Hence, more understandability and maintainability efforts are required. A high FOUT means that the class calls many other classes. However, the correlations presented by [101], [102], [103], [104] and [105] claim that FOUT does not affect software maintainability and understandability.

DIT measures the maximum length of a path from a class to a root class in the inheritance structure of a system. In fact, classes that are deep down in the classes' hierarchy potentially inherit many methods from super-classes, increasing the maintainability and understandability efforts. Hence, maintainability and understandability decline with increasing DIT. Moreover, deeper trees involve greater design complexity since more classes and methods are involved. On the other hand, the deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods. In Java programming language, where all classes inherit the class "Object", the minimum value of DIT is 1.

The NOC metric measures the number of direct sub-classes of a given class. It provides an indication of how a software reuses itself. Classes with a large number of children are considered difficult to modify and usually require greater maintainability and understandability effort since the effects of changes will propagate on to all children in the inheritance hierarchy. In addition, the greater the number of children, the greater the likelihood of improper abstraction of the parent class. Hence, a low NOC is desirable. NOC measures the breadth of a class hierarchy, whereas maximum DIT measures the depth.

7.3 Experiments and Results

All the experiments have been run on Windows 7 with Intel Core i5-2400 CPU. JHawk calculated the required software metrics for all subject systems. The calculation process was quite fast with JHawk taking only a few seconds to generate the results. Furthermore, based on a validated set of design pattern instances recovered using our research prototype, MLDA, two sets of classes were created: pattern classes and non-pattern classes.

7.3.1 Recovered Design Instances

Table 7.2 presents the number of design pattern instances implemented in all subject systems, as they are recovered by MLDA and as they are validated based on all publicly published results in the available literature. (i.e. the number of implemented design instances in a subject system is the number of true positive instances plus the number of false negative instances). Hence, all classes playing roles in design patterns were identified. As Table 7.2 demonstrates, 1051 design pattern instances were implemented in all subject systems. In addition, most creational, structural and behavioral instances were implemented in JRefactory, PMD and Nutch respectively. The Adapter and State/Strategy design patterns were implemented in all subject systems, whereas the interpreter design pattern was not implemented in any subject system. The maximum occurrence of a design pattern was for State/Strategy with 108 instances implemented in Nutch. Furthermore, as can be seen in Table 7.2, most implemented instances in PMD, MapperXML and Nutch are behavioral instances which require dynamic interactions between classes. Most JRefactory instances are creational instances which require initializations of classes.

Type	Design Patterns	JHotDraw	JRefactory	JUnit	QuickUML	Lexi	Nutch	PMD	MapperXML
Creational Patterns	Singleton	2	12	0	1	2	2	4	3
	Prototype	2	0	0	0	1	0	1	0
	Abstract factory	0	0	0	1	0	0	2	2
	Factory method	2	87	2	18	0	5	32	22
	Builder	0	2	0	1	0	0	16	1
Structural Patterns	Adapter	11	16	11	29	30	44	87	41
	Bridge	4	0	0	1	0	0	1	0
	Composite	1	0	1	1	3	0	0	1
	Decorator	3	1	1	3	0	4	0	0
	Façade	1	2	0	1	1	1	1	1
	Flyweight	1	0	0	1	1	1	0	1
	Proxy	0	0	0	2	0	4	4	0
Behavioural Patterns	ChainofResponsibility	0	0	0	0	0	0	1	0
	Command	9	25	0	18	6	25	3	26
	Interpreter	0	0	0	0	0	0	0	0
	Iterator	0	0	1	0	0	1	0	0
	Mediator	0	0	0	0	0	0	0	0
	Memento	0	0	0	0	10	1	1	12
	Observer	2	0	1	1	0	0	1	3
	State/Strategy	6	11	3	10	1	113	21	11
	Visitor	2	2	0	0	0	0	1	0
	Template method	4	4	1	3	0	7	108	56
	Total	**50**	**162**	**21**	**91**	**55**	**208**	**284**	**180**

Table 7.2: Total number of design instances implemented in subject systems

7.3.2 Participation Percentage in the Total Metric Value

After the identification of all design pattern instances in all subject systems, the number of classes playing roles in design patterns, the number of classes playing roles in structural, creational and behavioral instances and the number of classes playing more than one role have been identified.

Table 7.3 presents the total number of classes playing roles in design patterns for all subject systems. A class may play more than one role and participate in two or more different design patterns. The percentage of design patterns in a subject system is the percentage of classes playing roles in design patterns to the total number of system classes. Around 40% of the subject systems' classes participate and play roles in design patterns. Most

QuickUML classes play roles in design patterns, whereas only 28% of JHotDraw classes play roles.

Number of classes/ Subject Systems	**JHotDraw**	**JRefactory**	**JUnit**	**QuickUML**	**Lexi**	**Nutch**	**PMD**	**MapperXML**
Total number of classes	201	612	112	145	170	398	570	374
NCPR in creational instances	8	122	3	37	24	12	83	52
NCPR in structural instances	35	35	30	66	43	100	112	79
NCPR in behavioral instances	30	80	10	39	20	82	155	114
Classes playing more than one role	17	48	5	37	25	54	133	76
NCPR in all design pattern instances	56	189	38	105	62	140	217	169
Percentage of design pattern classes to the total system classes	**28%**	**31%**	**34%**	**72%**	**36%**	**35%**	**38%**	**45%**
Note: **NCPR**: Number of Classes Playing Roles								

Table 7.3: The number of classes playing roles in design patterns in all subject systems

Table 7.4 illustrates the percentage of participation in the total metric value for both pattern classes and non-pattern classes in all subject systems. Consistent behavior can be noticed for size and inheritance metrics. These metrics negatively affect the maintainability and understandability of a subject system. More specifically, the calculated averages for size and inheritance metrics indicate that the participation of pattern classes, in the total metric value, is less than that of the other classes in the system. Hence, total system size and inheritance were shaped based on the non-pattern classes.

On the other hand, the average participation of the pattern classes in the total coupling metrics, except RFC, is higher than that of the other classes in the system. Both groups, participate almost equally in the average RFC metric. The functionality of the system might be relying on the design pattern classes (i.e. design pattern classes provide key functionality to the system) which require interacting and collaboration between pattern classes and other classes in the system. This could explain why the participation of design pattern classes, in the total system coupling, is higher than that of other classes in the system. This is consistent with the results presented by [73] where design pattern implementations increased the coupling metrics on the class level for the relevant classes.

Furthermore, total system cohesion was formulated based on the non-pattern classes, which contradicts the common belief that the implementation of design pattern enhances system cohesion, as suggested in [61].

Consequently, the participation of pattern classes in five out of nine metrics is less than that of other classes in the system. These metrics have a negative impact on software maintainability and understandability. In contrast to common beliefs, design pattern classes shaped the system coupling. The pattern participant classes provide the key functionality to the system, which may explain why these classes tend to couple and interact with other classes. In addition, the total system cohesion was formulated based on the non-pattern classes. High values of cohesion are desirable.

Non-pattern classes implement more methods than do pattern classes in all subject systems, except in PMD and MapperXML. In addition, pattern classes of PMD and MapperXML have RFC, CBO and F-IN higher than non-pattern classes do. This could be explained by investigating the implemented design instances in these two systems where most of the implemented instances are behavioral instances. Behavioral instances are concerned with interaction between classes, which requires method calls and collaborations between classes. Hence, pattern classes tend to have more methods and coupling with other classes. Furthermore, in most subject systems, pattern classes have fewer cohesion values than do non-pattern classes. Consequently, total system cohesion relies on the non-pattern classes. This is a sign of the improper use of pattern classes where they perform more than a single purpose function.

Furthermore, inheritance metrics, depth of inheritance tree and number of children of pattern classes for all subject systems are all fewer than those of non-pattern classes. Hence, pattern classes require less maintainability and understandability effort.

Pattern classes of FOUT metric, which has been reported to have no impact on software maintainability and understandability, have fewer metric values than non-pattern classes. Consistent with other coupling metrics, pattern classes of PMD and MapperXML have fewer FOUT metric values than other classes in the system since most of their design instances are behavioral instances.

We noticed that total system coupling and cohesion was directly affected by the implementation of behavioral design instances. Classes playing roles and participating in behavioral patterns had higher coupling and lower cohesion than did other classes in the system.

System/ Metrics		NOM	LCOM	RFC	CBO	NLOC	F-IN	DIT	COH	FOUT	NOC
JHotDraw	pattern classes	50%	23%	50%	48%	44%	54%	30%	38%	42%	29%
	non-pattern classes	50%	77%	50%	52%	56%	46%	70%	62%	58%	71%
JRefactory	pattern classes	34%	23%	34%	35%	31%	40%	20%	30%	32%	24%
	non-pattern classes	66%	67%	66%	65%	69%	60%	80%	70%	68%	76%
JUnit	pattern classes	26%	24%	26%	42%	23%	60%	14%	32%	28%	13%
	non-pattern classes	74%	76%	74%	58%	77%	40%	86%	68%	72%	87%
QuickUML	pattern classes	47%	22%	48%	58%	48%	66%	31%	32%	51%	69%
	non-pattern classes	53%	78%	52%	42%	52%	34%	69%	68%	49%	31%
Lexi	pattern classes	45%	21%	45%	46%	41%	66%	12%	8%	25%	50%
	non-pattern classes	55%	79%	55%	54%	59%	34%	88%	92%	75%	50%
Nutch	pattern classes	40%	13%	41%	55%	38%	68%	21%	20%	43%	NA
	non-pattern classes	60%	87%	59%	45%	62%	32%	79%	80%	57%	NA
PMD	pattern classes	70%	35%	70%	66%	67%	80%	44%	35%	60%	13%
	non-pattern classes	30%	65%	30%	34%	33%	20%	56%	65%	40%	73%
MapperXML	pattern classes	66%	64%	67%	75%	66%	76%	56%	56%	76%	19%
	non-pattern classes	34%	36%	33%	25%	34%	24%	44%	44%	24%	81%
Average	pattern classes	**47%**	**29%**	**48%**	**53%**	**45%**	**64%**	**28%**	**31%**	**45%**	**33%**
	non-pattern classes	**53%**	**71%**	**52%**	**47%**	**55%**	**36%**	**72%**	**69%**	**55%**	**67%**

Note:
NOM: Number of Methods **LOC**: Lines Of Codes **RFC**: Response For Class **CBO**: Coupling Between Objects
LCOM: Lack of Cohesion in Methods **COH**: Cohesion **F-IN**: Fan-IN **FOUT**: Fan-out
DIT: Depth of Inheritance Tree **NOC**: Number of Children

Table 7.4: The participation percentage in the total metric value for both classes groups in all subject systems

7.4 Threats to Validity

There are possible threats to the validity of the presented results in this chapter. To correlate the internal software metrics to the external quality attributes, the correlations presented by previous studies, [101], [102], [103], [104] and [105], have been used Hence, our findings are subject to the significance of the previous correlations.

In addition, the calculated software metrics rely on the capabilities of the JHawk tool. Using other tools may show fewer differences in the calculated metrics. Finally, the design pattern occurrences and classes playing roles were recovered using our research prototype MLDA. To ensure more accurate validation, the recovered instances have been validated based on all publicly available results and repositories in the literature. Hence, using other tools to recover the design pattern instances may show some differences in terms of design instances and pattern classes.

7.5 Summary

This chapter presented a metrics-based approach to address the impact of design pattern instances on software maintainability and understandability. This approach classifies systems classes into two groups: classes that are playing roles in design patterns (pattern classes) and classes that are not playing roles in design patterns (non-pattern classes). Size, coupling and inheritance metrics were calculated for both groups using JHawk, a Java metrics tool. Furthermore, the participation percentage (for both groups) in the total metric value was calculated in attempts to address the degree of participation of pattern classes in the total system metric value. The correlation of metrics to software maintainability and understandability presented by previous research studies has been used.

The experiment results illustrate that design pattern classes have fewer roles in size and inheritance metrics than do non-pattern classes, a sign that design pattern classes enhance software understandability and maintainability. On the other hand, non-pattern classes have fewer roles in coupling metrics than do pattern classes, which contradicts the common beliefs that design patterns enhance system coupling and cohesion. Hence, no safe conclusion can be drawn regarding the impact of design patterns on software maintainability and understandability.

The only conclusion that can be drawn is that design pattern classes have better size and inheritance metrics than non-pattern classes do. In addition, design pattern classes provide the key functionality of the system, which may explain why these classes tend to interact and why they are coupled with other classes in the system.

However, further investigations are required to reach a safe conclusion. One possible solution is to apply certain refactoring techniques to create two versions of a system: patterns and non-patterns versions. Then software metrics can be calculated and compared with both versions.

Chapter Eight

Conclusions and Future Directions

This chapter concludes the thesis with a summary of the conducted work, revisits the research questions and presents the contribution of this thesis. The limitations of the work and future research directions are presented at the end of this chapter.

8.1 Conclusions

Design patterns have a key role in the software development process. They describe both the structure and behavior of classes and their relationships. Design patterns can improve software documentation, speed up the development process and enable large-scale reuse of software architectures.

This thesis presented a hybrid structural rule-based approach to recover GoF design patterns from the Java source code. In addition, a metrics-based approach has been presented in attempts to investigate the impact of design pattern instances on software understandability and maintainability.

The main research questions that this thesis aims to address are:

1. Is the Structural Search Model (SSM), which relies on the relationships matching, able to recover instances of design patterns with a reasonable detection accuracy?
2. Is the use of a rule-based system to match the method signatures of the candidate design instances to that of the subject system able to reduce the number of false positive candidate instances (i.e. enhancing the detection accuracy of design patterns, which relies on relationships matching)?

3. Do classes playing roles in design patterns have better software metrics than other classes in the system (i.e. do design pattern instances enhance certain software quality attributes)?

Concerning the first research question, SSM produces too many false positive instances. Hence, the structure of design patterns is not enough to recover their instances from the Java source code. Although SSM relies on five key relationships, uses a class-level representation of the subject system, builds the design pattern structure incrementally and tries all possible combinations of classes until a complete pattern structure is achieved, it produces too many false positive instances. Moreover, SSM is not able to recover instances of the Template design patterns since this pattern relies on one relationship (the inheritance relationship) and method recovery capabilities were required to recover its instances.

For the second research question, we tried to enhance the detection accuracy of SSM by developing a rule-based system which matches the method signatures of the candidate design instances to that of the subject system. The rule-based system represented the method signatures of the candidate design instances as a set of rules, whereas the method signatures of the subject system are represented as a set of facts. The rules and facts have been created using the so-called MLDA rules/facts generator. Rules that represent the true positive instances should be fired. On the other hand, rules that represent the false negative instances should not be fired since they have the structure of design patterns and have failed to implement the required method signatures. The CLIPS inference engine has been used to match the set of rules and facts.

As the experiments demonstrate, most of the false positive instances have been removed. Representing the method signatures using a rule-based approach provides a fingerprint for design patterns inside the source code. Both the structure and the method signatures are required to detect accurately the instances of GoF design patterns. The rule-based system enhances the detection accuracy of design patterns, which relies on the relationship matching.

Concerning the third research question, the metrics-based approach could not reach a safe conclusion regarding the impact of GoF design patterns on software understandability and maintainability. This approach classifies system classes into two groups: classes that are playing roles and participating in design patterns (pattern classes) and classes that are not playing roles or participating in design patterns (non-pattern classes). Specifically, a list of classes playing roles in design patterns was generated, based on our validated recovered design instances and using our research prototype MLDA, from eight subject systems. This motivates us to build the metrics based approach. The participation percentage in the total metric value has been calculated for both groups.

However, the metrics-based approach shows that classes that play roles in design patterns have better inheritance and size metrics than do non-pattern classes. This gives a sign that design patterns enhance software understandability and maintainability. The total system inheritance and size metrics rely on the pattern classes. In contrast, non-pattern classes have better coupling metrics than do pattern classes where the total system coupling relies on the pattern classes, contradicting the common belief that design patterns enhance system coupling.

The key observations can be summarized as follows:

1. Design patterns can vary in their implementations inside the source code. To overcome this challenge, this thesis uses the standard format presented by GoF.
2. There is no agreed reference benchmark to validate the recovered design pattern instances. In fact, we refer to all publicly available results and repositories in the literature. Consequently, an MLDA repository for design pattern instances in eight subject systems has been developed. This repository can be used by other researchers in the future to validate their recovered design instances.

3. The matching of the method signatures of the candidate design instances to that of the subject system using a rule-based system enhances the detection accuracy of design patterns, which relies on relationship matching. But relationship matching produces too many false positive instances and is not enough to recover design pattern instances.
4. Classifying the system classes into pattern classes and non-pattern classes and calculating the participating percentage in the total metric value shows that pattern classes shape the total system inheritance and size and fail to play fewer roles than do non-pattern classes in the coupling metrics. This is mainly due to the fact that design patterns provide the key functionality to the system, which requires more interaction and coupling with other classes.

This thesis has added to the body of software engineering real evidence that the applying of a rule-based approach enhances the detection accuracy of GoF design patterns, which relies on relationships matching. In addition, this thesis shows that classes playing roles in design patterns shape the total system inheritance and size metrics.

8.2 Limitations

There are some limitations of the work presented in this thesis. First of all, design pattern instances are recovered based on the standard structural format presented by GoF. Secondly, the way in which the results are validated could affect precision and recall. To validate the number of true positives, false positives and false negatives, we refer to all publicly published results in the available literature.

Thirdly, this thesis focuses on Java programming language. It could be worthwhile to conduct the evaluation on other projects having different languages.

Finally, the proposed metrics-based approach relies on previous studies to correlate the internal software metrics to the external quality attributes. Hence, our findings, regarding the impact of GoF design patterns, are subject to the significance of the previous correlations.

8.3 Future Directions

MLDA relies on static analysis capabilities to recover the instances of design patterns. A dynamic analysis level could be added to record the dynamic behavior of pattern participant classes and record the messages interaction during the runtime. Thus, all false positive instances could be eliminated.

Furthermore, since MLDA uses the standard format of design patterns, SSM could be enhanced to recover different variants of design pattern. This can be done by summarizing all the possible variants of each design pattern and adding variant structures as possible candidate structures on to SSM. Moreover, SSM would allow the user to build its own pattern by adding the number of required relationships for each pattern.

Based on the validated set of design pattern instances recovered from eight subject systems, the MLDA benchmark for design pattern instances, a tool for automatic validation of recovered design instances could be developed. This tool would validate the design pattern instances recovered by other detection tools. In addition, the validation tool could inform the researcher whether the recovered instance was a true or a false positive and show a list of all false negative instances. Moreover, the researcher could update the repository by entering its validated set of design pattern instances recovered from other subject systems.

Finally, the metrics based approach could be used to compare the metrics set of two system versions: the pattern version and the non-pattern version. However, producing two identical versions for the same system would not be an easy task since both versions should provide the same functionality. Certain refactoring techniques should be applied to ensure that both versions provide the same functionality. Another option would be to track versions of a software and to compare their metrics set. This could offer a sign of the impact of design patterns.

References

[1] Rayl, A.J.S. (October 16, 2008). "NASA Engineers and Scientists-Transforming Dreams Into Reality". http://www.nasa.gov/index.html. NASA. Retrieved December 27, 2014.

[2] Pressman, R.S.,"Software Engineering: A Practitioner's Approach", McGraw-Hill, 2010.

[3] IEEE Computer Society and the ACM. “Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering”, 2004.

[4] W. H. Thomas, "Fundamentals of Digital Computer Programming”, Proceedings of the I.R.E., IEEE Computer Society, 1953.

[5] D. N. Reps, G. J. Kirk, JR., "Distribution System Primary-Feeder Voltage Control II-Digital Computer Program", Transactions of the American Institute of Electrical Engineers Power Apparatus and Systems, Part III, Vol 77, Issue 3, IEEE Computer Society, 1958.

[6] L. B. Lusted, "Computer Programming of Diagnostic Tests", IRE Transactions on Medical Electronics", Vol ME-7, Issue 4, IEEE Computer Society, 1960

[7] Len Bass, Paul Clements, and Rick Kazman. Software Architecture in Practice, 3rd Edition, Addison-Wesley Professional, 2012.

[8] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J. Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2003.

[9] Chikofsky, E.J., and J.H. Cross. II, “Reverse Engineering and Design Recovery: A Taxonomy,” IEEE Software, January 1990, pp. 13-17.

[10] Alexander, C., Ishikawa, S., Silverstein, M., 1977. A Pattern Language: Town, Buildings, Construction. Oxford University Press, New York.

[11] Gamma, E., Helms, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading, MA.

[12] McCall, J., P. Richard, and G. Walters, “Factors in Software Quality,” three volumes, NTIS AD-A049-014,015,055, November 1977.

[13] Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., and Merritt, M., Characteristics of Software Quality, North Holland, 1978.

[14] ISO, International Organization for Standardization, "ISO 9000:2000, Quality management systems - Fundamentals and vocabulary", 2000.

[15] Grady, R. B., Practical software metrics for project management and process improvement, Prentice Hall, 1992.

[16] Kruchten, P., the Rational Unified Process an Introduction – Third Edition, Addison Wesley Longman, Inc., 2003.

[17] Dromey, R.G., "Concerning the Chimera [software quality]", IEEE Software, no. 1, pp. 33-34, 1996.

[18] G. Rasool, I. Philippow, P. Mader, “Design Pattern Recovery Based on Annotations”. International Journal of advances in Engineering Software, Vol 41, Issue 4, 2010, pp. 519-526.

[19] K. Stencel, and P. Wegrzynowicz, “Detection of Diverse Design Pattern Variants”, 15th Asia-Pacific Software Engineering Conference, 2008, pp. 25-32.

[20] M. Vokac, “ An efficient tool for recovering design patterns from C++ code”, Journal of Object Technology, Volume 5, No. 1, 2006, pp. 139–157.

[21] Scientific Tool works Inc. Understand for C++, 2003.

[22] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page; Pattern based reverse-engineering of design components. In ICSE 99: Proceedings of the 21st International Conference on Software Engineering, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[23] Paakki J., Karhinen A., Gustafsson J., Nenonen L. and Verkamo A.I., Software metrics by architectural pattern mining, Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), 2000, 325–332.

[24] Niere, J., Shafer, W., Wadsack, J.P., Wendehals, L., Walsh, J., 2002. Towards pattern design recovery. In: Proceedings of International Conference on Software Engineering (ICSE’02), Orlando, FL, USA, pp. 338–348.

[25] G. Antoniol, R. Fiutem, and L. Cristoforetti, “Design pattern recovery in object-oriented software”, In Proceedings of the 6th international workshop on program comprehension, 1998, pp. 153–160.

[26] M. V. Detten, and S. Becker, “Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems”, In

Proceedings of the 7th International Conference on the Quality of Software Architectures, QoSA, pp. 23-32, 2011.

[27] Uchiyama, S., Kubo, A., Washizaki, H., and Fukazawa, Y. (2014). Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning. Journal of Software Engineering and Applications, 7, 983-998. doi: 10.4236/jsea.2014.712086.

[28] Jochen Seemann and Juergen Wolff von Gudenberg. Pattern-based design recovery of java software. In SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, pages 10–16, New York, NY, USA, 1998. ACM Press.

[29] Felix Agustin Castro Espinoza, Gustavo Nuez Esquer, and Joel Surez Cansino. Automatic design patterns identification of C++ programs. In EurAsia-ICT 02: Proceedings of the First EurAsian Conference on Information and Communication Technology, pages 816–823, London, UK, 2002. Springer-Verlag.

[30] Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus—reverse engineering tool a schema for C++. 18th IEEE international conference on software maintenance (ICSM'02), pp. 172–181, October 2002.

[31] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.: Design Pattern Detection Using Similarity Scoring. IEEE Transaction on Software Engineering 32(11) (2006).

[32] Dong, J., Lad, D.S., Zhao, Y.: Dp-miner: Design pattern discovery using matrix. In: Proc. 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2007, pp. 371–380 (2007).

[33] Dongjin Yu, Yanyan Zhang, and Zhenli Chen: A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. The Journal of Systems and Software 103 (2015) 1–16.

[34] Ampatzoglou, A., Michou, O., Stamelos, I., 2013b. Building and mining a repository of design pattern instances: practical and research benefits. EntertainmentComput.4, 131–142.

[35] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Working Conference on Reverse Engineering, pages 208–1996.

[36] Y.-G. Guéhéneuc and N. Jussien, "Using Explanations for Design Patterns Identification," Proc. First IJCAI Workshop Modelling and Solving Problems with Constraints, C. Bessie`re, ed., pp. 57-64, Aug. 2001.

[37] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, “Fingerprinting Design Patterns,” Proc. 11th Working Conf. Reverse Eng. (WCRE’04), Nov. 2004.

[38] Beyer, D., Lewerentz, C. 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).

[39] J. McC. Smith, and D. Stotts. SPQR: Flexible Automated Design Pattern Extraction from Source Code. In Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, Montreal QC, Canada, October, 2003, pp. 215-224.

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

[41] Y.-G. Guéhéneuc and G. Antoniol, “DeMIMA: A Multi-Layered Framework for Design Pattern Identification,” IEEE Trans. Software Eng., vol. 34, no. 5, pp. 667-684, Sept./Oct. 2008.

[42] Lucia, A.D., Deufemia, V., Gravino, C., and Risi, M., Design pattern recovery through visual language parsing and source code analysis, The Journal of Systems and Software, Vol 82, pp. 1177–1193, 2009.

[43] M. Zanoni, “Data mining techniques for design pattern detection,” Ph.D. dissertation, Universita degli Studi di Milano-Bicocca, 2012.

[44] Alnusair, A., Zhao, T., Yan, G., 2014. Rule based detection of design patterns in program code. Int.J.Softw.ToolsTechnol.Trans.16 (3), 315–334.

[45] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk, Master’s thesis, North Carolina State University, 1996.

[46] Bansiya Jagdish: Automating Design-Pattern Identification. Dr. Dobb’s Journal. June 1998.

[47] Kim, H. and Boldyreff, C. (2000) A Method to Recover Design Patterns Using Software Product Metrics. In Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability, Vienna, 27-29 June 2000, 318-335.

[48] Heuzeroth, D., Holl, T., Hogstrom, G., Lowe, W., 2003. Automatic design pattern detection. In: Proceedings of International Workshop on Program Comprehension (IWPC’03), Portland, OR, USA, pp. 94–103.

[49] Philippow, I., Streitferdt, D., Riebish, M., Naumann, S., 2005. An approach for reverse engineering of design patterns. Software System Modeling 4 (1), 55–79.

[50] A. Blewitt. Hedgehog: Automatic Verification of Design Patterns in Java. PhD thesis, School of Informatics, University of Edinburgh, 2005. http://www.bandlem.com/Alex/Papers/PhDThesis.pdf.

[51] Kaczor O., Guéhéneuc Y-G, Hamel S. Efficient identification of design patterns with bit-vector algorithm. In Proceedings of the 10th European conference on software maintenance and reengineering, Bari, Italy; 22–24 March 2006. p. 184–93.

[52] W.B Frakes and R.Baeza, Yates, Information Retrieval: Data Structure and Algorithms, Prentice Hall, 1992.

[53] Danny B. Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In Proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications, pages 342 – 357. ACM Press, 1995.

[54] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and J¨urgenEbert, editors, Proceedings of 5th Conference on Software Maintenance and Reengineering, pages 77–84. IEEE Computer Society Press, March 2001.

[55] Wydaeghe, K. Verschaeve, B. Michiels, B. Van Damme, E. Arckens, and V. Jonckers: Building an OMT-editor using design patterns: An experience report, 1998.

[56] William B. McNatt and James M. Bieman. Coupling of design patterns: Common practices and their benefits. In T.H. Tse, editor, Proceedings of the 25th Computer Software and Applications Conference, pages 574–579. IEEE Computer SocietyPress, October 2001.

[57] B. Ellis, J. Stylos and B. Myers, “The Factory Pattern in API Design: A Usability Evaluation”, Proceedings of the 29th international conference on Software Engineering, IEEE, pp. 302–312, Minneapolis, Minnesota, 20–26 May 2007.

[58] J. Hannemann and G. Kiczales. “Design Pattern Implementation in Java and AspectJ”, Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA ‘02), ACM, pp. 161–173, Seattle, Washington, 4–8 November 2002.

[59] S. Jeanmart, Y.-G. Guéhéneuc, H. Sahraoui and N. Habra, “A Study of the Impact of the Visitor Design Pattern on Program Comprehension and Maintenance Tasks”, Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM’ 09), IEEE, pp. 69–78, Lake Buena Vista, Florida,15–16 October 2009.

[60] B.A. Malloy and J.F. Power, “Exploiting design patterns to automate validation of class invariants: Research articles”, Software Testing Verification & Reliability, Wiley Interscience, 16 (2), pp. 71–95, June 2006.

[61] A. Ampatzoglou and A. Chatzigeorgiou, “Evaluation of object oriented design patterns in game development”, Information and Software Technology, Elsevier, 49 (5), pp. 445–454, May 2007.

[62] L. Aversano, G. Canfora, L. Cerulo, C. D. Grosso and M. Di Penta, “An empirical study on the evolution of design patterns”, Foundations of Software Engineering (FSE’ 07), ACM, pp. 385–394, Dubrovnik, Croatia, 3–7 September 2007.

[63] J. M. Bieman, G. Straw, H. Wang, P. W. Munger and R. T. Alexander, “Design Patterns and Change Proneness: An Examination of Five Evolving Systems”, Proceedings of the 9th International Symposium on Software Metrics, IEEE, pp. 40, Sydney, Australia, 03–05 September 2003.

[64] M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc and G. Antoniol, “An empirical study of the relationships between design pattern roles and class change proneness”, Proceedings of the IEEE International Conference on Software Maintenance (ICSM’08), IEEE, pp. 217–226, Beijing, China, 28 September–04 October 2008.

[65] M. Gatrell, S. Counsell and T. Hall, “Design Patterns and Change Proneness: A Replication Using Proprietary C# Software”, Proceedings of the 2009 16th Working Conference on Reverse Engineering ,pp. 160–164, Lille, France, 13–16 October 2009.

[66] B. Baudry, Y. Le Sunye and J. M. Jezequel, “Toward a ‘Safe’ Use of Design Patterns to Improve OO Software Testability”, Proceedings of the 12th International Symposium on Software Reliability Engineering, IEEE, pp. 324, Hong Kong, China, 27–30 November 2001.

[67] B. Baudry, Y. Le Traon, G. Sunye and J. M. Jezequel, “Measuring and Improving Design Patterns Testability”, Proceedings of the 9th International Symposium on Software Metrics, IEEE, pp. 50, Sydney, Australia, 03–05 September 2003.

[68] M. Elish, “Do Structural Design Patterns Promote Design Stability?, Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC’06), IEEE, pp 215–220, Chicago, Illinois, 17–21 September 2006.

[69] Design Patterns in Java: Reference and Example Site, Online at http://www.fluffycat.com/java/patterns.html.

[70] F. Khomh and Y.-G. Guéhéneuc, “Do Design Patterns Impact Software Quality Positively?”, Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering, IEEE, pp. 274–278, Athens, Greece, 01–04 April 2008.

[71] L. Prechelt, B. Unger-Lamprecht, W.F. Tichy, P. Brossler and L. G. Votta, “A controlled experiment in maintenance comparing design patterns to simpler solutions”, IEEE Transactions on Software Engineering, IEEE, 27 (3), pp. 1134–1144, December 2001.

[72] M. Vokac, W. Tichy, D. I. K. Sjoberg, E. Arisholm and M.Aldrin, “A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns—A Replication in a Real Programming Environment”, Empirical Software Engineering, Springer, 9(3), pp 149–195, September 2004.

[73] K. Kouskouras, A. Chatzigeorgiou and G. Stephanides, “Facilitating software extension with design patterns and Aspect-Oriented Programming”, Journal of Systems and Software, Elsevier, 81 (10), pp 1725–1737, October 2008.

[74] H. Rajan, S. M. kautz and W. Rowcliffe, “Concurrency by Modularity: Design Patterns, a Case in Point”, Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA ‘10), ACM, pp. 790–805, Reno, Nevada, 17–21 October 2010.

[75] Arcelli Fontana, F., Caracciolo, A., Zanoni, M., 2012. DPB: A benchmark for design pattern detection tools. In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR’12). IEEE Computer Society, Szeged, Hungary, pp. 235–244. doi:10.1109/C.

[76] Y.-G. Guéhéneuc, “P-MARt: Pattern-like micro architecture repository,” Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, 2007.

[77] Fülöp, L. J., Hegedus, P., & Ferenc, R. (2008). BEFRIEND - A benchmark for evaluating reverse engineering tools. Periodica Polytechnica, Electrical Engineering, 52(3-4), 153-162. DOI: 10.3311/pp.ee.2008-3-4.04.

[78] CLIPS: A Tool for Building Expert Systems. 2016. downloads. [ONLINE] Available at: http://www.clipsrules.net/. [Accessed 5 January 2017].

[79] Frost, R., Hafiz, R. and Callaghan, P. (2008) “Parser Combinators for Ambiguous Left-Recursive Grammars." 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN, Volume 4902/2008, Pages: 167 - 181, January 2008, San Francisco.

[80] Github. 2012. JavaParser by javaparser. [ONLINE] Available at: https://javaparser.github.io/javaparser/. [Accessed 1 March 2015].

[81] Herbert Schildt (2014) Java: The Complete Reference, 9th edition edn., : McGraw-Hill Osborne.

[82] Paul J. Deitel and Harvey M. Deitel (2014) Java How To Program (Early Objects), 10th edition edn., : Pearson.

[83] Cay S. Horstmann and Gary Cornell (2012) Core Java Volume I--Fundamentals, 9th edition edn., : Prentice Hall.

[84] Dane Cameron (2014) Java 8: The Fundamentals, 1st edition edn.,: Cisdal Publishing.

[85] Cay S. Horstmann (2015) Core Java for the Impatient, 1st edition edn.,: Addison-Wesley Professional.

[86] Cay S. Horstmann (2012) Big Java: Early Objects, 5th edition edn.,: John Wiley & Sons Inc.

[87] Erich Gamma and Thomas Eggenschwiler. JHotDraw start page (no date). Available at: http://www.jhotdraw.org/ (Accessed: 10 February 2016).

[88] Mike Atkinson. (2003). JRefactory. Available at: http://jrefactory.sourceforge.net/. (Accessed: 20 February 2015).

[89] Erich Gamma, Kent Beck, David Saff and Mike Clark. (2004). JUnit. Available at: http://junit.org/junit4/.

[90] Crahen E., Alphonce C., Ventura P. (2002), “QuickUML: A beginner’s UML tool”, in OOPSLA ’02, Seattle, Washington, USA.

[91] Brill Pappin and Matthew Schmidt. (1999). Lexi - Java based Word Processor. Available at: http://lexi.sourceforge.net/.

[92] M. Phelan, [online] Available: http://mapper.sourceforge.net/.

[93] Apache Software Foundation. Stable release (2015). Apache Nutch. Available at: http://nutch.apache.org/.

[94] PMD. Stable release (2017). PMD. Available at: https://pmd.github.io/.

[95] Ligęza, A.: Logical Foundations for Rule-based Systems, 2nd edn. Springer, Heidelberg (2006).

[96] Forgy, Charles L. "Rete: A fast algorithm for the many pattern/many object pattern match problem." Artificial intelligence 19.1 (1982): 17-37.

[97] Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag, Germany.

[98] JHawk Metrics Tool, http://www.virtualmachinery.com/jhawkprod.htm, 2017.

[99] Mark Lorenz and Jeff Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994.

[100] S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object-Oriented Design", IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June 1994.

[101] Dandashi, F.: A Method for Assessing the Reusability of Object-Oriented Code Using a Validated Set of Automated Measurements. In: Proceedings of the ACM Symposium on Applied Computing, pp. 997–1003 (2002).

[102] Bruntink, M.; Deursen, A.: An empirical study into class testability. J. Syst. Softw. 79, 1219–1232 (2006).

[103] Y.Kanellopoulos, P.Antonellis, D. Antoniou, C.Makris, E.Theodoridis, C. Tjortjis and N.Tsirakis, "Code Quality Evaluation Methodology Using The Iso/Iec 9126 Standard," International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.3, July, 2010, pp.17-36.

[104] Mohammed Alshayeb: The Impact of Refactoring to patterns on Software Quality Attributes. Arab J Sci Eng. 36: 1241-1251 (2011).

[105] D. Spinellis, Code Quality-The Open Source Perspective, Addison-Wesley, 2006.

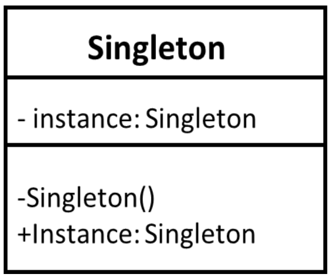
A Appendix

Design Patterns Library Generated by MLDA

Note that Class diagrams as they presented by GoF.

A. Creational Design Patterns

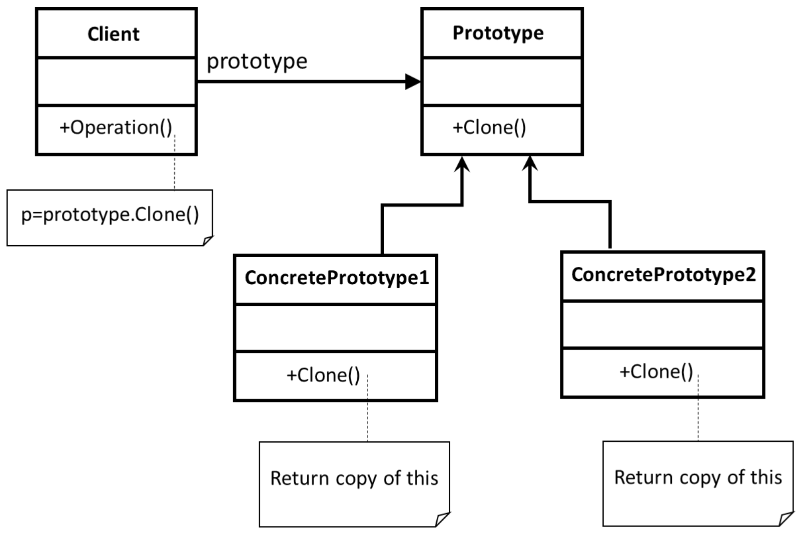
1. **Singleton**



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Singleton	Singleton	ASSOCIATION

Stats	
Relation_Type	**Count**
ASSOCIATION	1

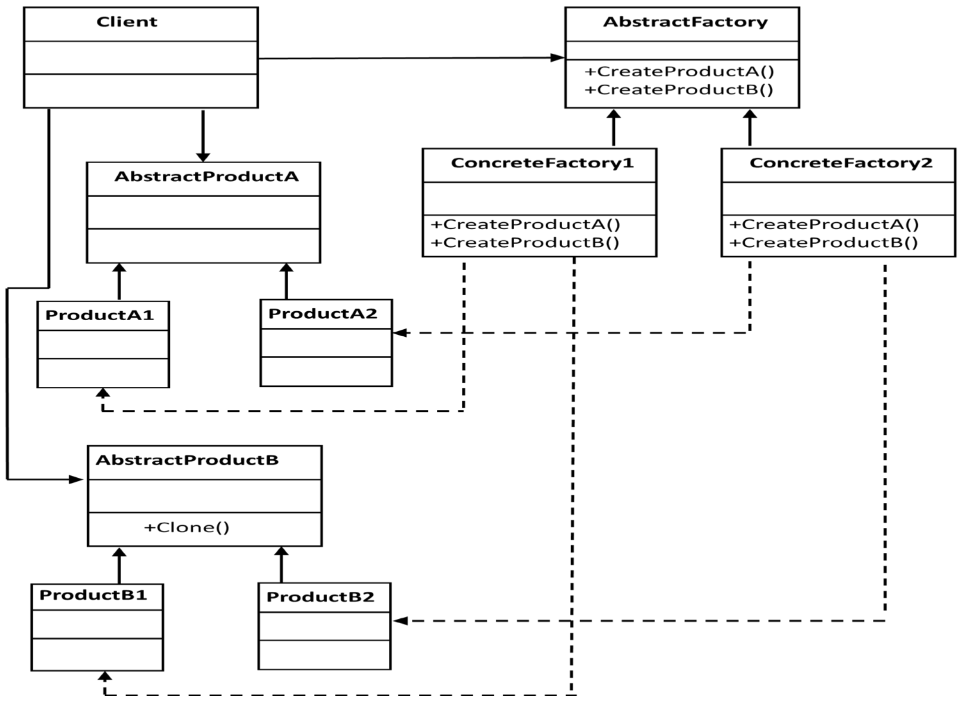
2. **Prototype**



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Client	Prototype	AGGREGATION
ConcretePrototype1	Prototype	DEPENDENCY
ConcretePrototype1	Prototype	REALIZATION
ConcretePrototype2	Prototype	DEPENDENCY
ConcretePrototype2	Prototype	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	2
AGGREGATION	1
DEPENDENCY	2

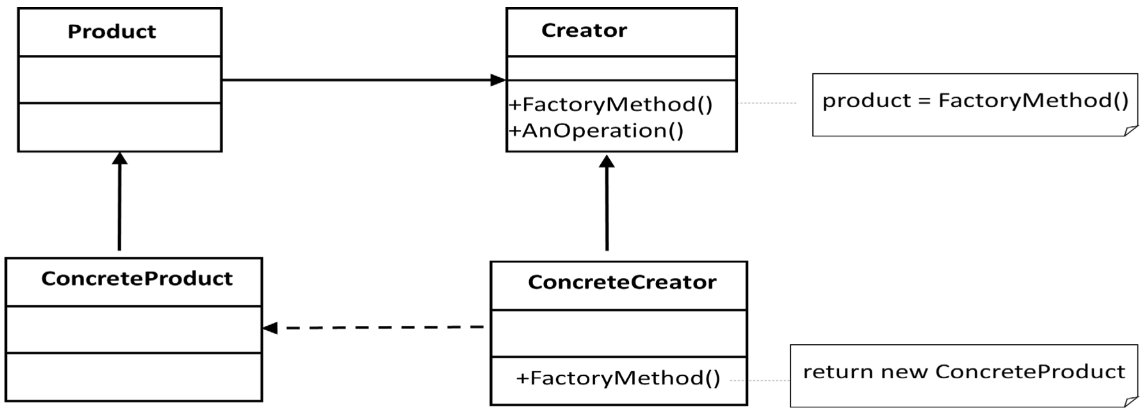
3. Abstract Factory



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Client	AbstractProductA	AGGREGATION
Client	AbstractProductB	AGGREGATION
ConcreteFactory1	AbstractFactory	REALIZATION
ConcreteFactory2	AbstractFactory	REALIZATION
ConcreteFactory1	ProductA1	ASSOCIATION
ConcreteFactory1	ProductB1	ASSOCIATION
ConcreteFactory2	ProductA2	ASSOCIATION
ConcreteFactory2	ProductB2	ASSOCIATION
ProductA1	AbstractProductA	REALIZATION
ProductA2	AbstractProductA	REALIZATION
ProductB2	AbstractProductB	REALIZATION
ProductB1	AbstractProductB	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	6
AGGREGATION	2
ASSOCIATION	4

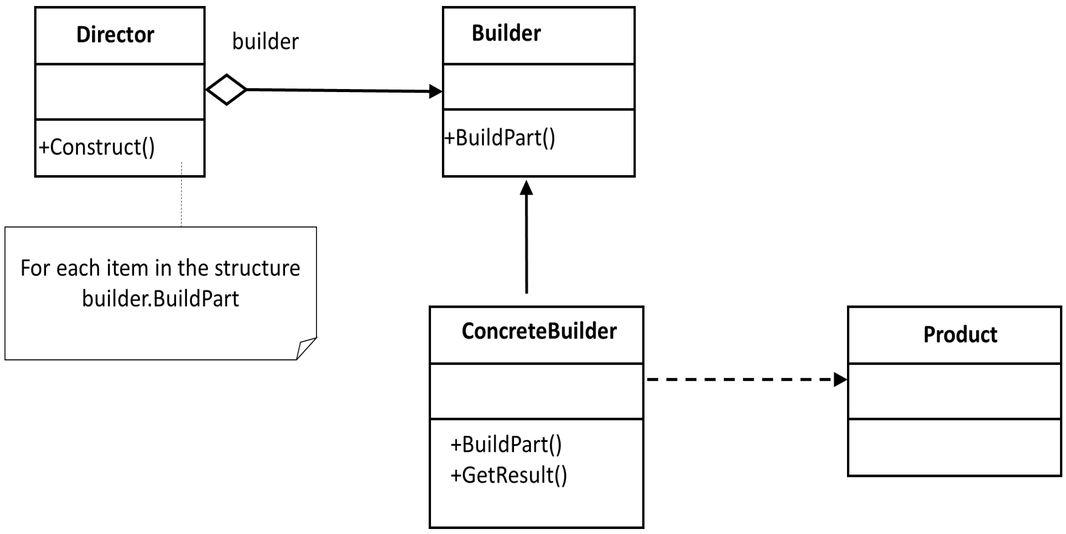
4. Factory



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
ConcreteProduct	Product	REALIZATION
ConcreteCreator	Creator	INHERITANCE
ConcreteCreator	ConcreteProduct	AGGREGATION

Stats	
Relation_Type	**Count**
INHERITANCE	1
AGGREGATION	1
REALIZATION	1

5. Builder

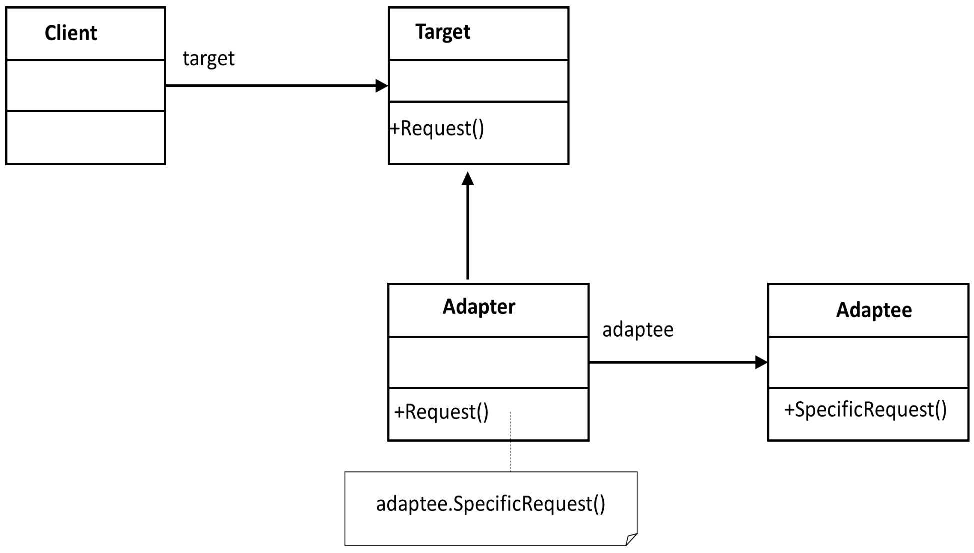


Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Director	Builder	ASSOCIATION
ConcreteBuilder	Product	ASSOCIATION
ConcreteBuilder	Builder	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	1
ASSOCIATION	2

B. Structural Design Patterns

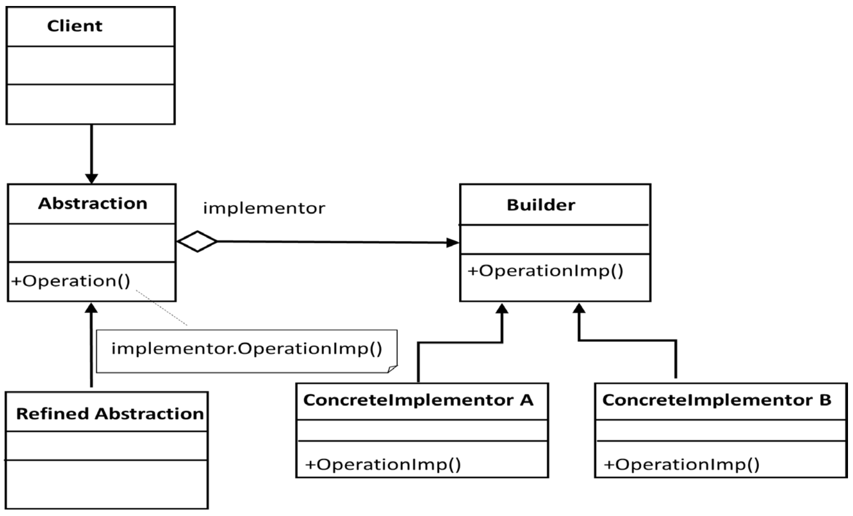
6. Adapter



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Adapter	Adaptee	ASSOCIATION
Adapter	Target	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	1
ASSOCIATION	1

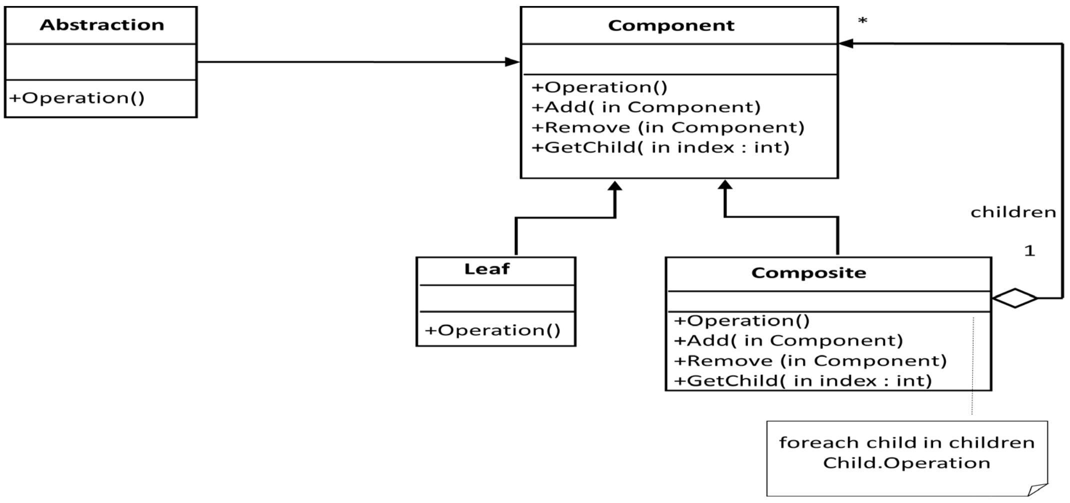
7. Bridge



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Abstraction	Implementor	AGGREGATION
ConcreteImplementorA	Implementor	REALIZATION
ConcreteImplementorB	Implementor	REALIZATION
RefinedAbstraction	Abstraction	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	3
AGGREGATION	1

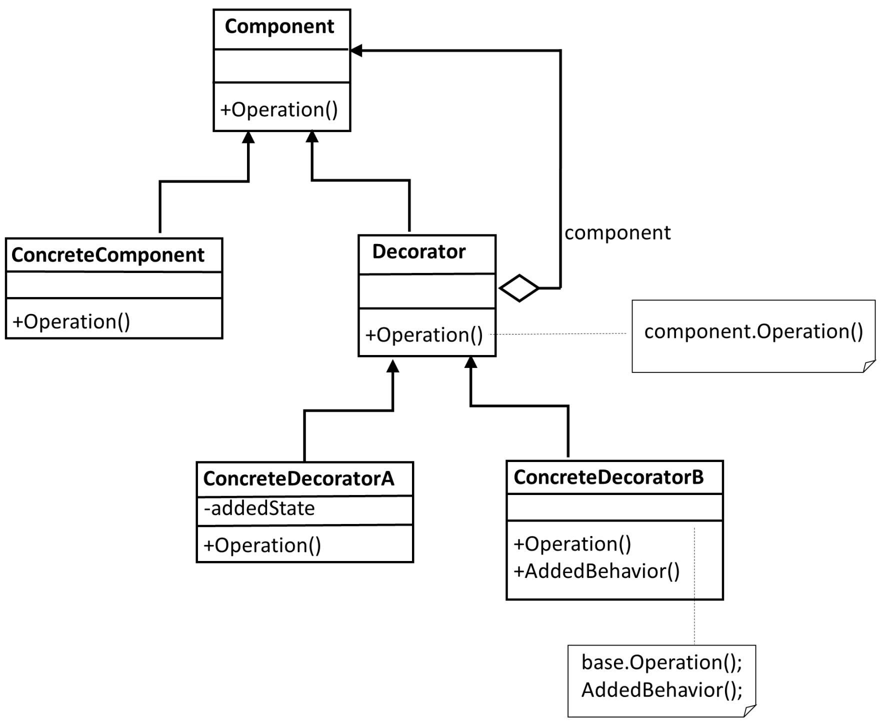
8. Composite



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Composite	Component	ASSOCIATION
Composite	Component	REALIZATION
Leaf	Component	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	2
ASSOCIATION	1

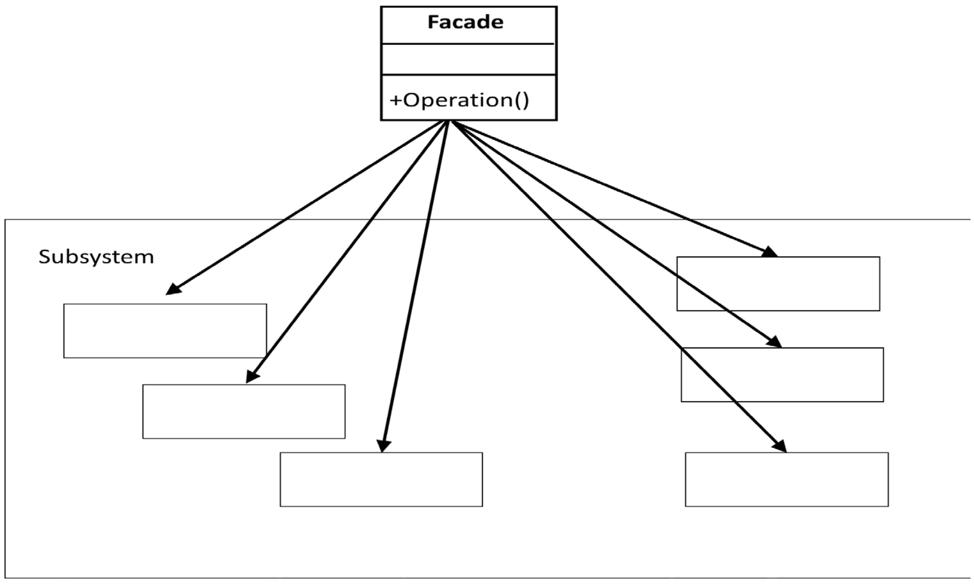
9. Decorator



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Decorator	Component	AGGREGATION
Decorator	Component	REALIZATION
ConcreteComponent	Component	REALIZATION
ConcreteDecoratorA	Decorator	REALIZATION
ConcreteDecoratorB	Decorator	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	4
AGGREGATION	1

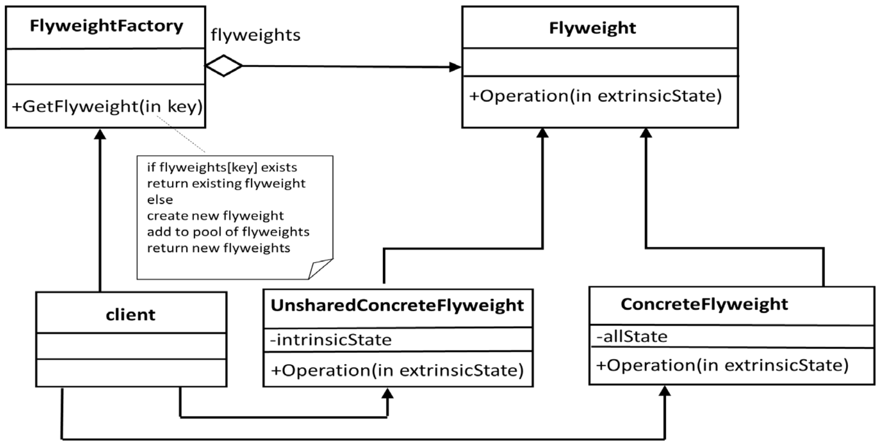
10. Facade



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Façade	SubSystemTwo	AGGREGATION
Façade	SubSystemThree	AGGREGATION
Façade	SubSystemOne	AGGREGATION

Stats	
Relation_Type	**Count**
AGGREGATION	3

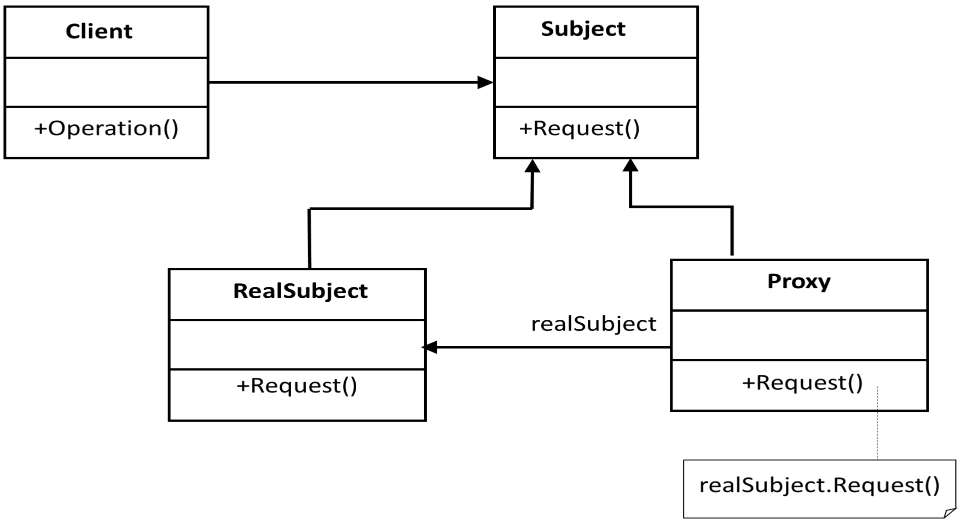
11. Flyweight


FlyweightFactory
+GetFlyweight(in key)
flyweights
Flyweight
+Operation(in extrinsicState)
if flyweights[key] exists
return existing flyweight
else
create new flyweight
add to pool of flyweights
return new flyweights
client
UnsharedConcreteFlyweight
-intrinsicState
+Operation(in extrinsicState)
ConcreteFlyweight
-allState
+Operation(in extrinsicState)

Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
FlyweightFactory	Flyweight	ASSOCIATION
UnsharedConcreteFlyweight	Flyweight	INHERITANCE
ConcreteFlyweight	Flyweight	INHERITANCE

Stats	
Relation_Type	**Count**
INHERITANCE	2
ASSOCIATION	1

12. Proxy

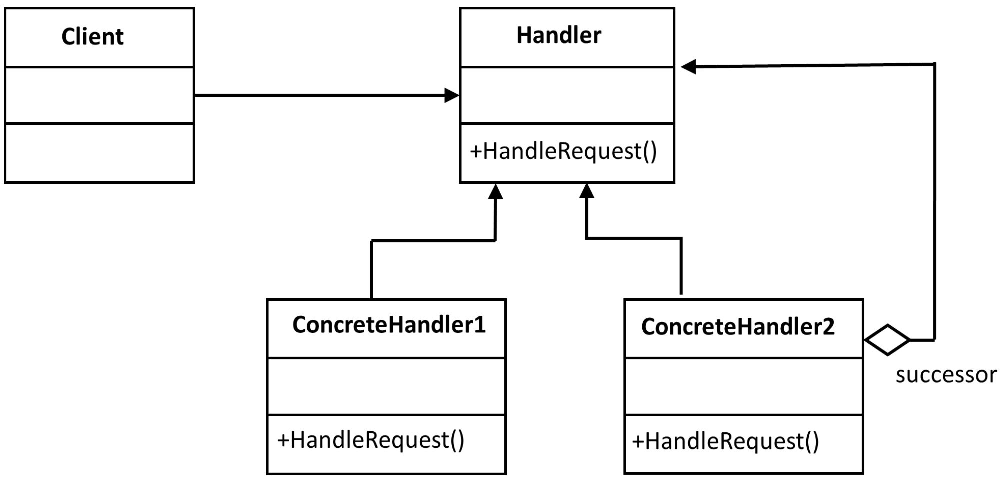


Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Proxy	Subject	REALIZATION
Proxy	RealSubject	ASSOCIATION
RealSubject	Subject	REALIZATION

Stats	
Relation_Type	**Count**
REALIZATION	2
ASSOCIATION	1

C. Behavioral Design Pattern

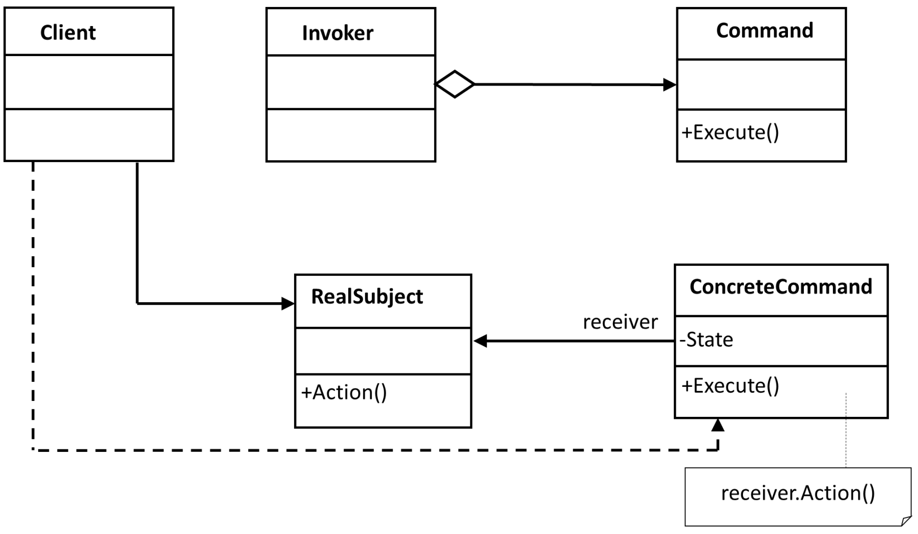
13. Chain of Responsibility



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Handler	Handler	AGGREGATION
ConcreteHandler1	Handler	REALIZATION
ConcreteHandler2	Handler	REALIZATION

Stats	
Relation_Type	**Count**
AGGREGATION	1
REALIZATION	2

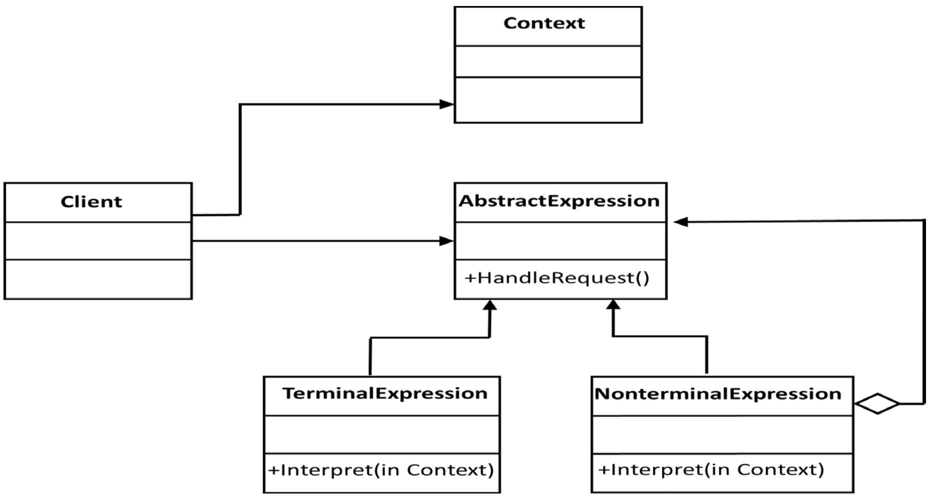
14. Command



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Invoker	Command	AGGREGATION
ConcreteCommand	Command	REALIZATION
Invoker	Command	AGGREGATION

Stats	
Relation_Type	**Count**
AGGREGATION	2
REALIZATION	1

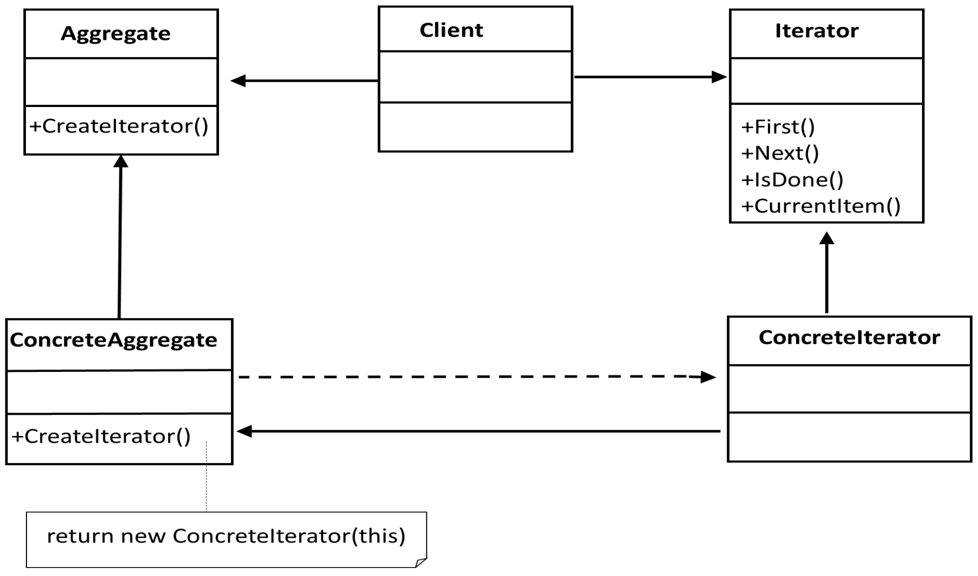
15. Interpreter



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
NonterminalExpression	AbstractExpression	REALIZATION
TerminalExpression	AbstractExpression	REALIZATION
NonterminalExpression	AbstractExpression	AGGREGATION
AbstractExpression	Context	AGGREGATION

Stats	
Relation_Type	**Count**
REALIZATION	2
AGGREGATION	2

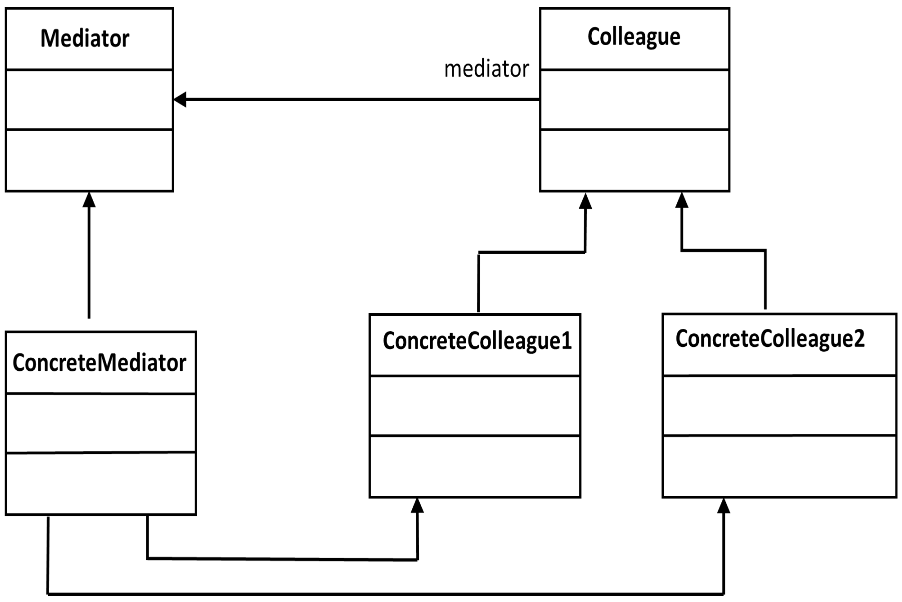
16. Iterator



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
ConcreteAggregate	Aggregate	REALIZATION
ConcreteIterator	ConcreteAggregate	AGGREGATION
ConcreteIterator	Iterator	REALIZATION
ConcreteAggregate	ConcreteIterator	ASSOCIATION

Stats	
Relation_Type	**Count**
ASSOCIATION	1
AGGREGATION	1
REALIZATION	2

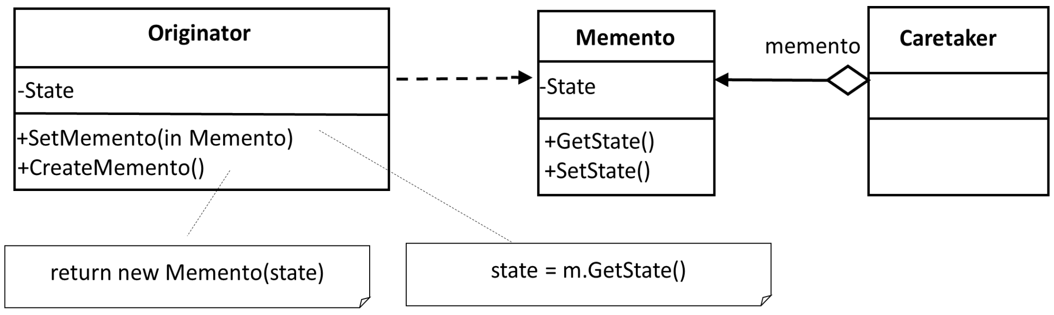
17. Mediator



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Colleague	Mediator	AGGREGATION
ConcreteColleague1	Colleague	INHERITANCE
ConcreteColleague2	Colleague	INHERITANCE
ConcreteMediator	ConcreteColleague2	AGGREGATION
ConcreteMediator	Mediator	REALIZATION
ConcreteMediator	ConcreteColleague1	AGGREGATION

Stats	
Relation_Type	**Count**
AGGREGATION	3
INHERITANCE	2
REALIZATION	1

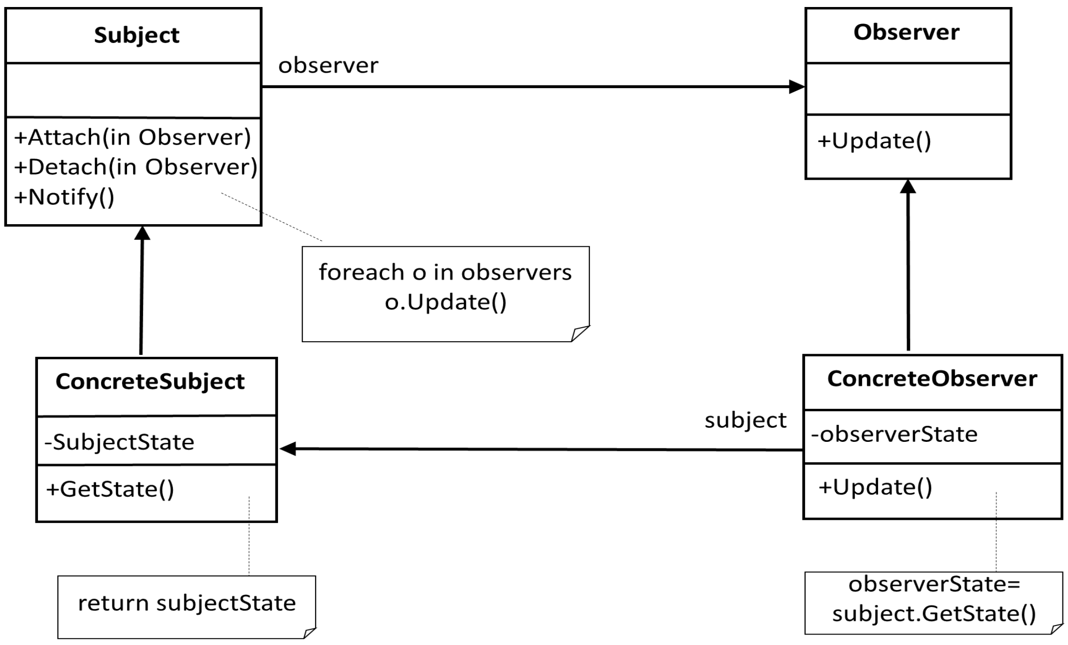
18. Memento



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Caretaker	Memento	AGGREGATION
Originator	Memento	ASSOCIATION

Stats	
Relation_Type	**Count**
AGGREGATION	1
ASSOCIATION	1

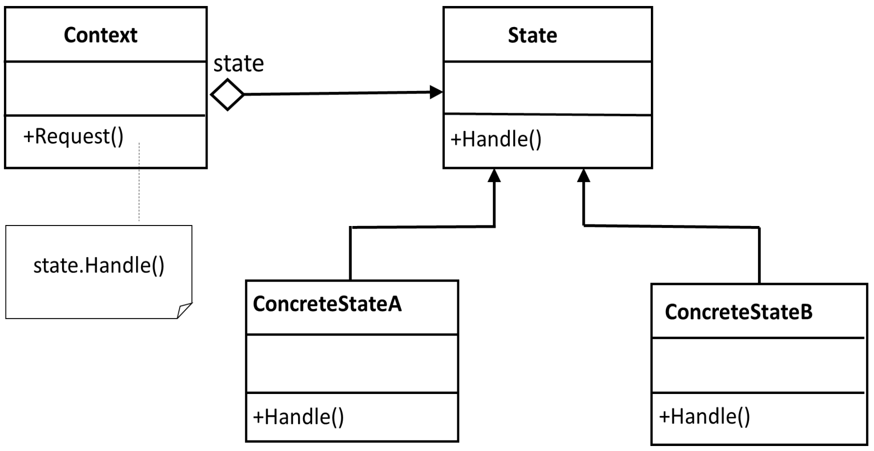
19. Observer



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Subject	Observer	ASSOCIATION
ConcreteObserver	Observer	REALIZATION
ConcreteObserver	ConcreteSubject	AGGREGATION
ConcreteSubject	Subject	REALIZATION

Stats	
Relation_Type	**Count**
ASSOCIATION	1
AGGREGATION	1
REALIZATION	2

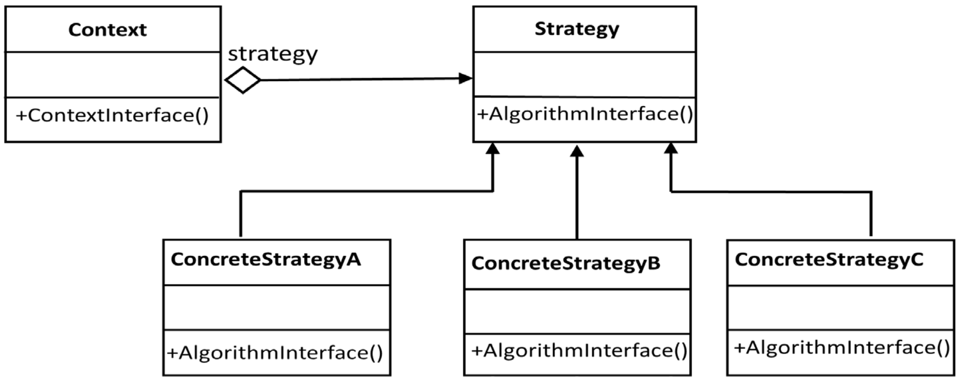
20. State



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Context	State	AGGREGATION
ConcreteStateA	State	REALIZATION
ConcreteStateB	State	REALIZATION

Stats	
Relation_Type	**Count**
AGGREGATION	1
REALIZATION	2

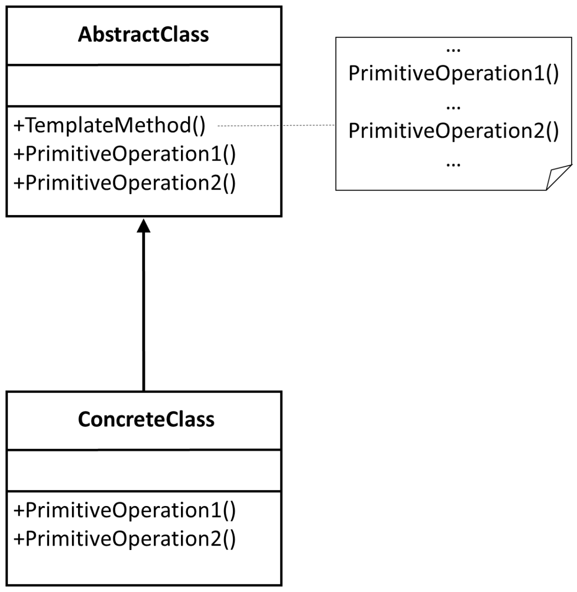
21. Strategy



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
Context	Strategy	AGGREGATION
ConcreteStrategyA	Strategy	REALIZATION
ConcreteStrategyB	Strategy	REALIZATION
ConcreteStrategyC	Strategy	REALIZATION

Stats	
Relation_Type	**Count**
AGGREGATION	1
REALIZATION	3

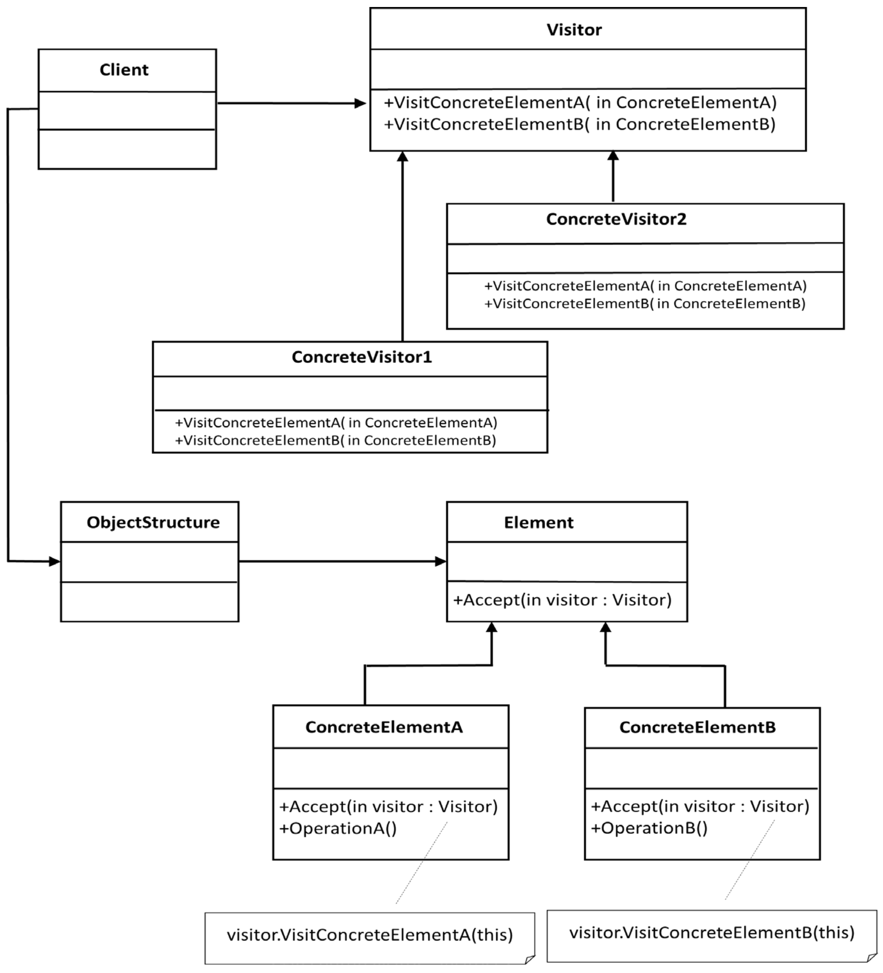
22. Template method



Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
ConcreteClassA	AbstractClass	INHERITANCE
ConcreteClassB	AbstractClass	INHERITANCE

Stats	
Relation_Type	**Count**
INHERITANCE	2

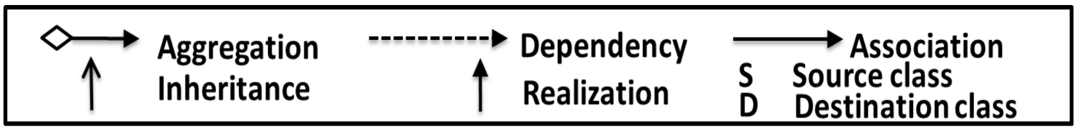
23. Visitor


Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
Client
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

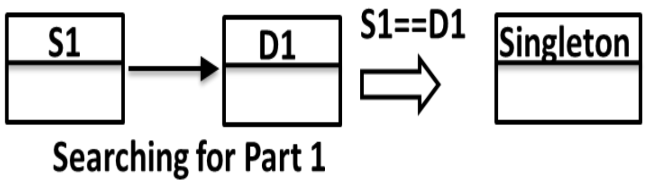
Class_Relations		
Source_Class	**Destination_Class**	**Relation_Type**
ObjectStructure	Element	ASSOCIATION
ConcreteElementA	Element	REALIZATION
ConcreteElementB	Element	REALIZATION
ConcreteVisitor1	Visitor	REALIZATION
ConcreteVisitor2	Visitor	REALIZATION
ConcreteVisitor1	ConcreteElementA	AGGREGATION
ConcreteVisitor2	ConcreteElementB	AGGREGATION

Stats	
Relation_Type	**Count**
ASSOCIATION	1
AGGREGATION	2
REALIZATION	4

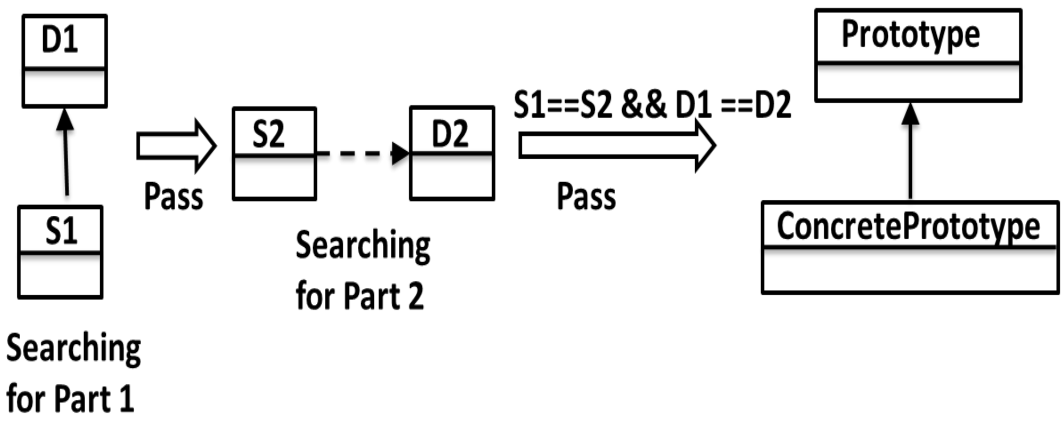
B Appendix
Structural Search Model (SSM)



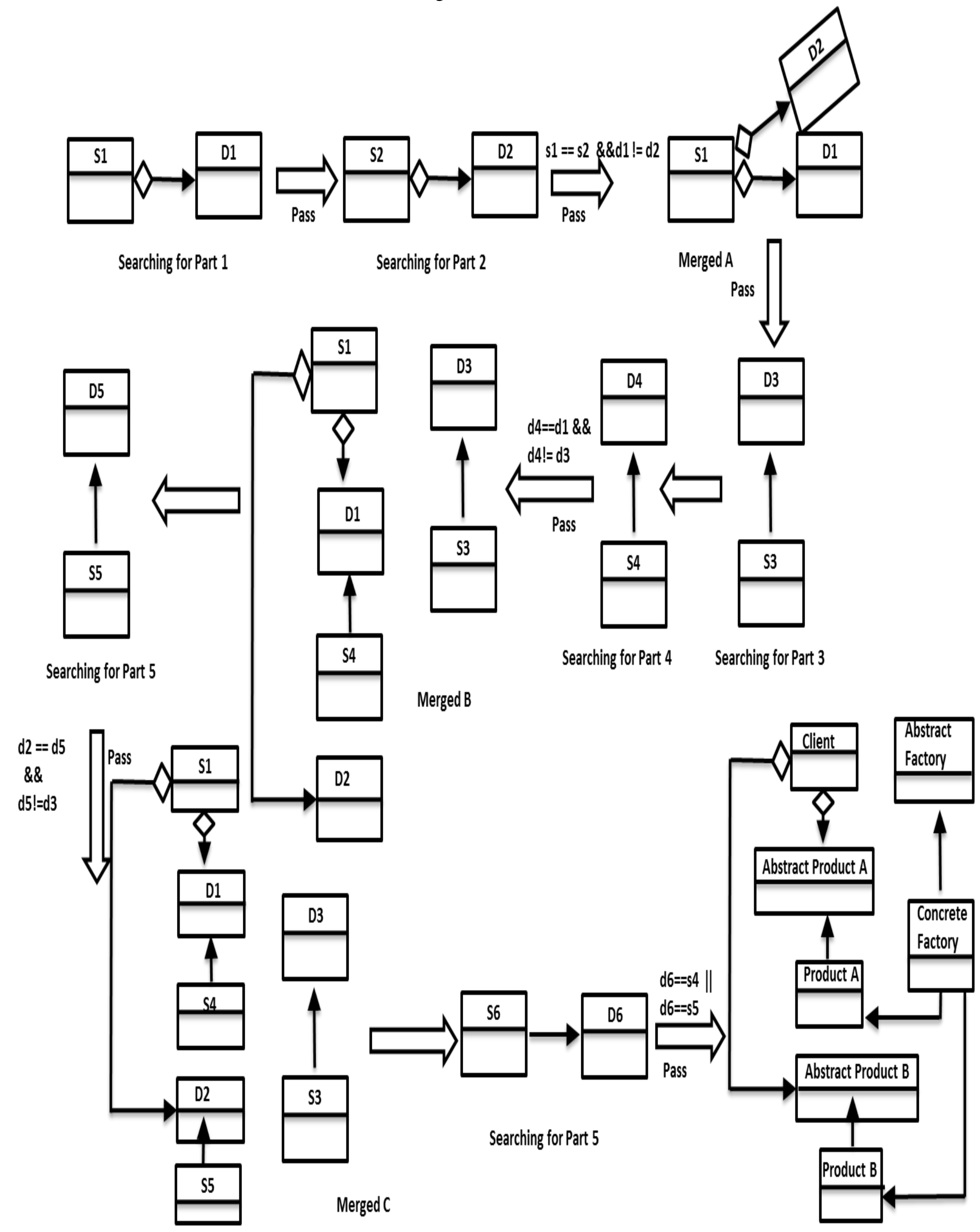
1. Singleton



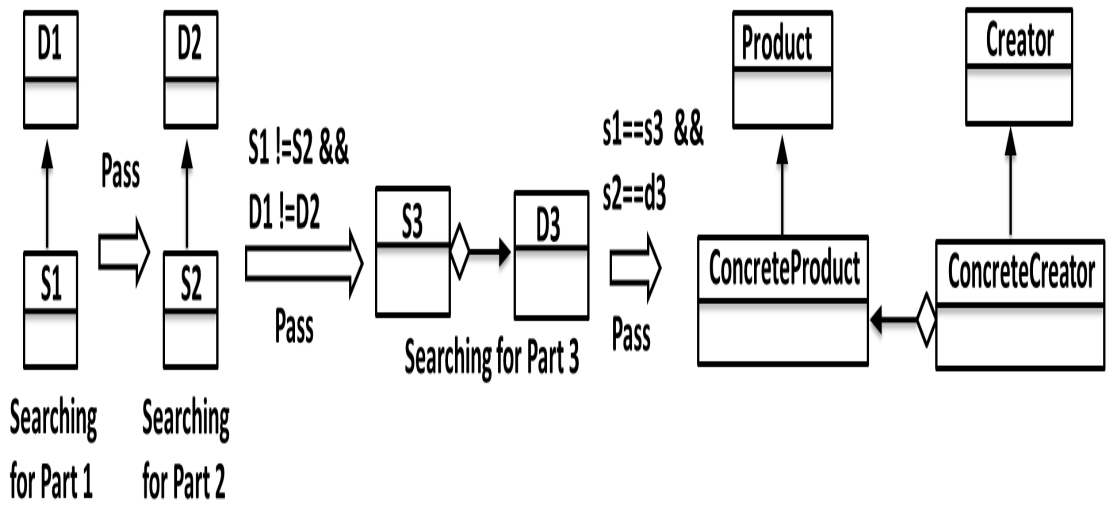
2. Prototype



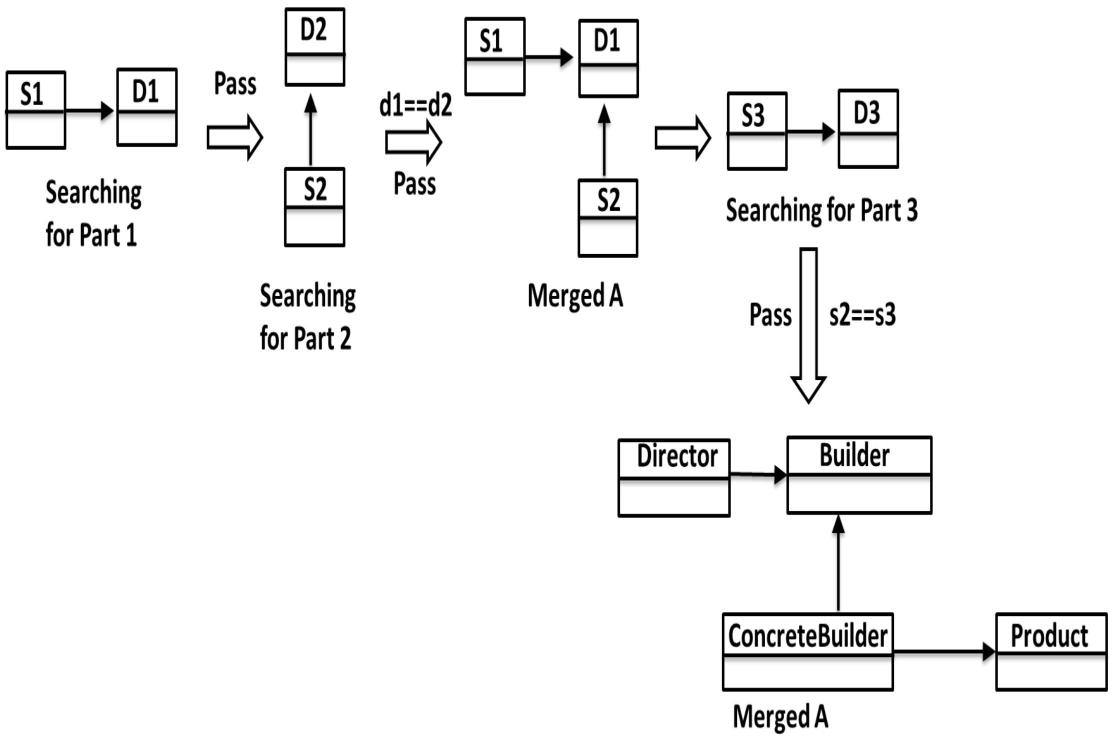
3. Abstract Factory



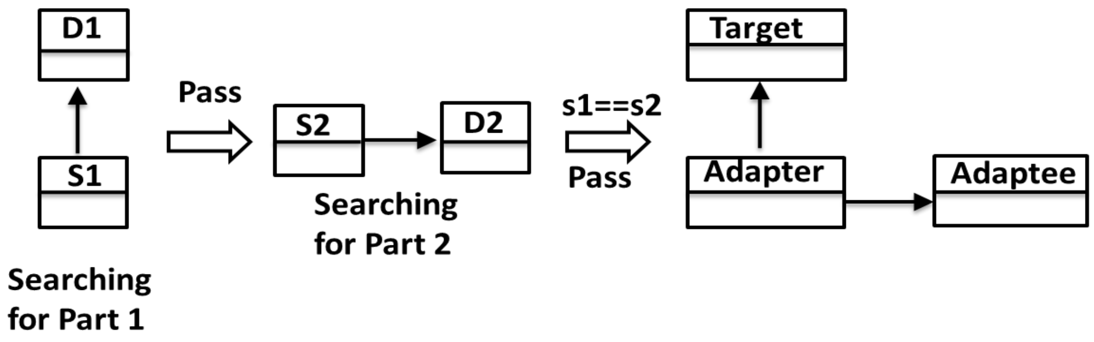
4. Factory



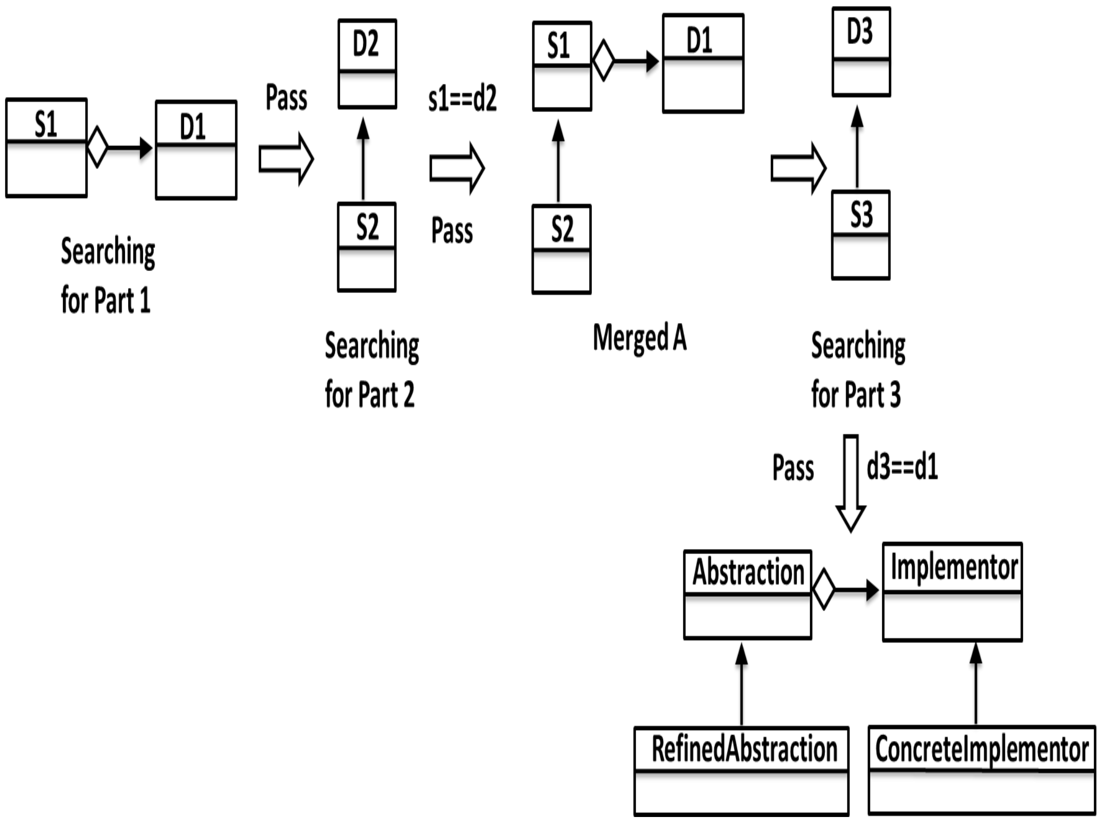
5. Builder



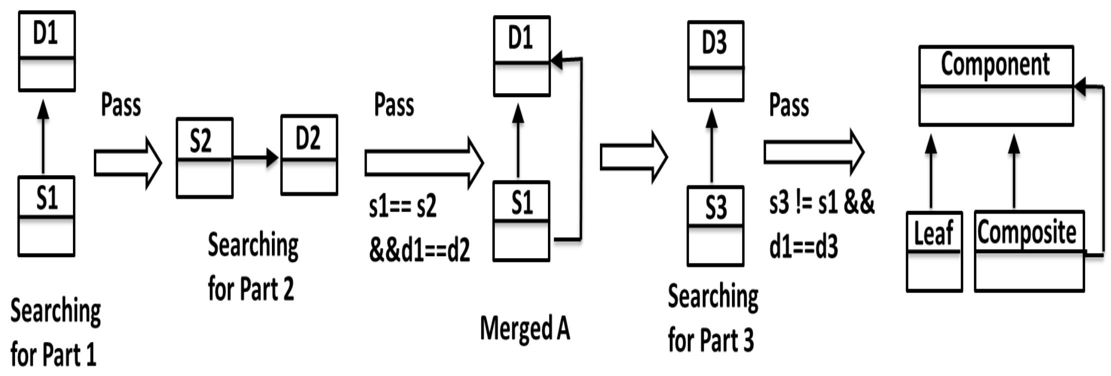
6. Adapter



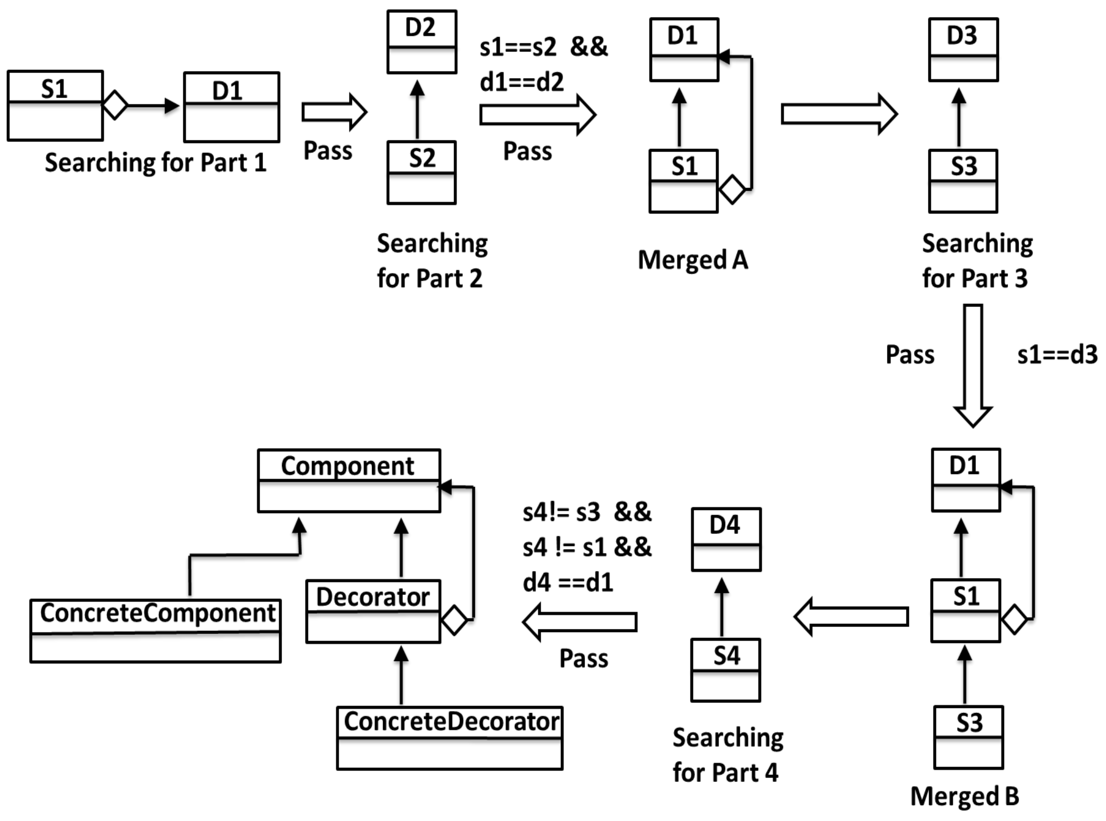
7. Bridge



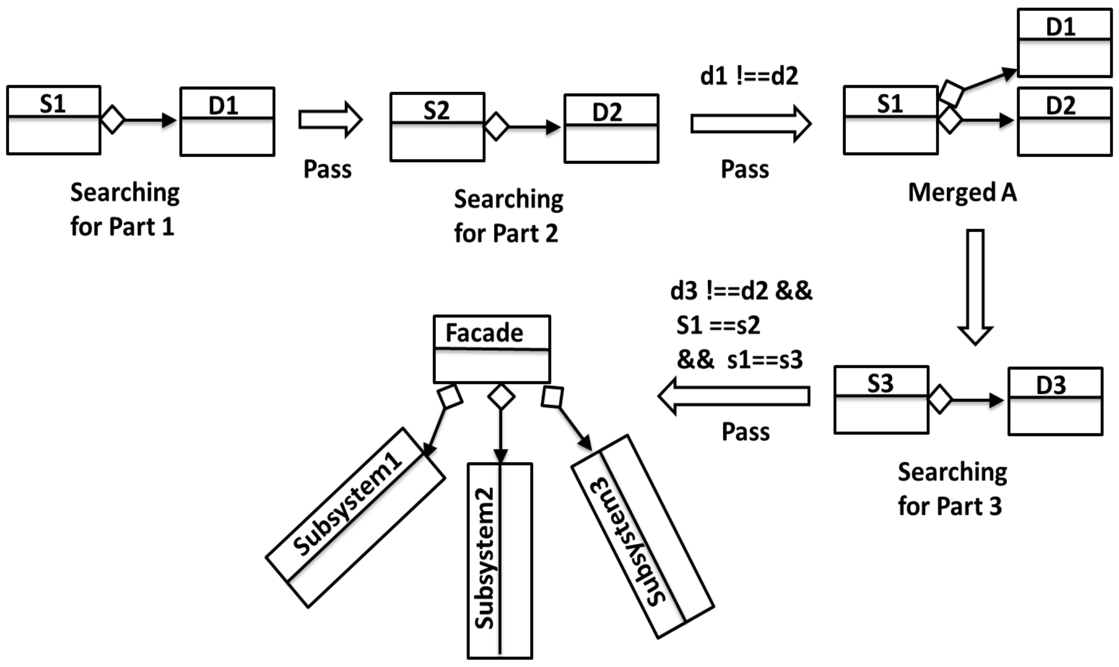
8. Composite



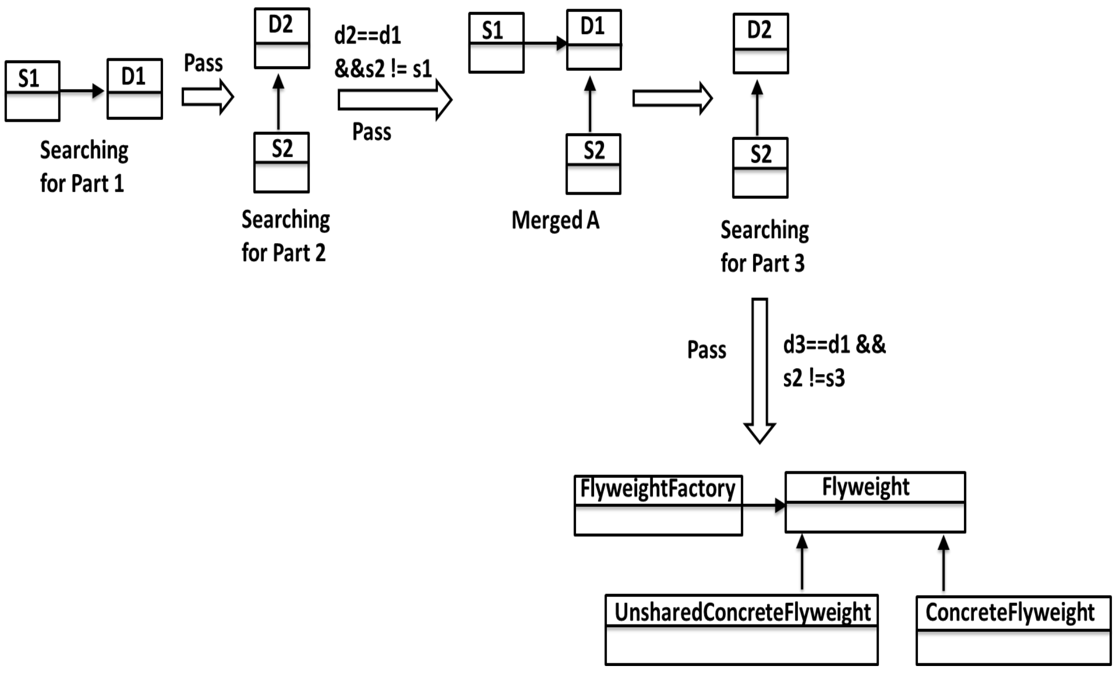
9. Decorator



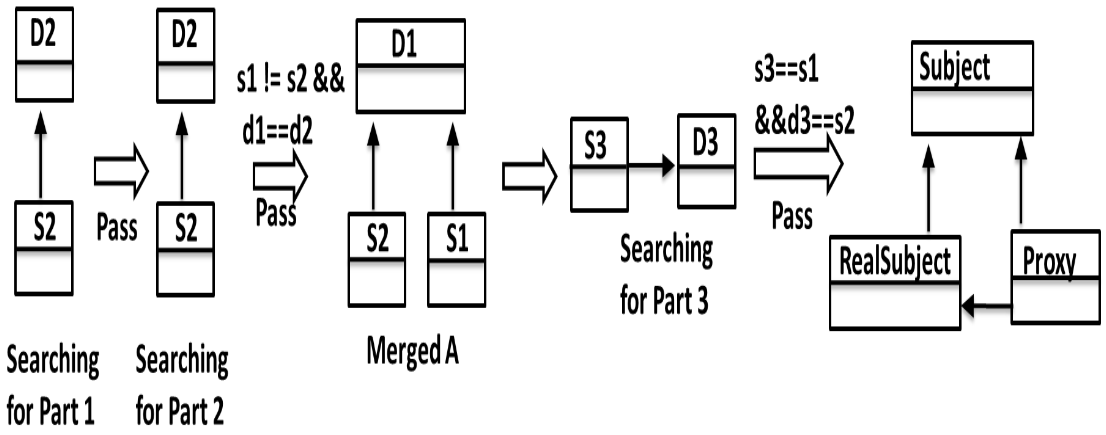
10. Façade



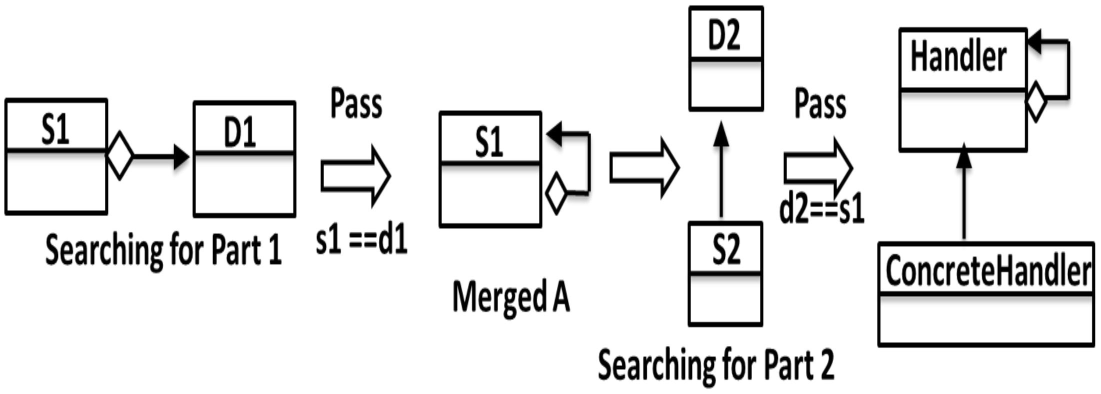
11. Flyweight



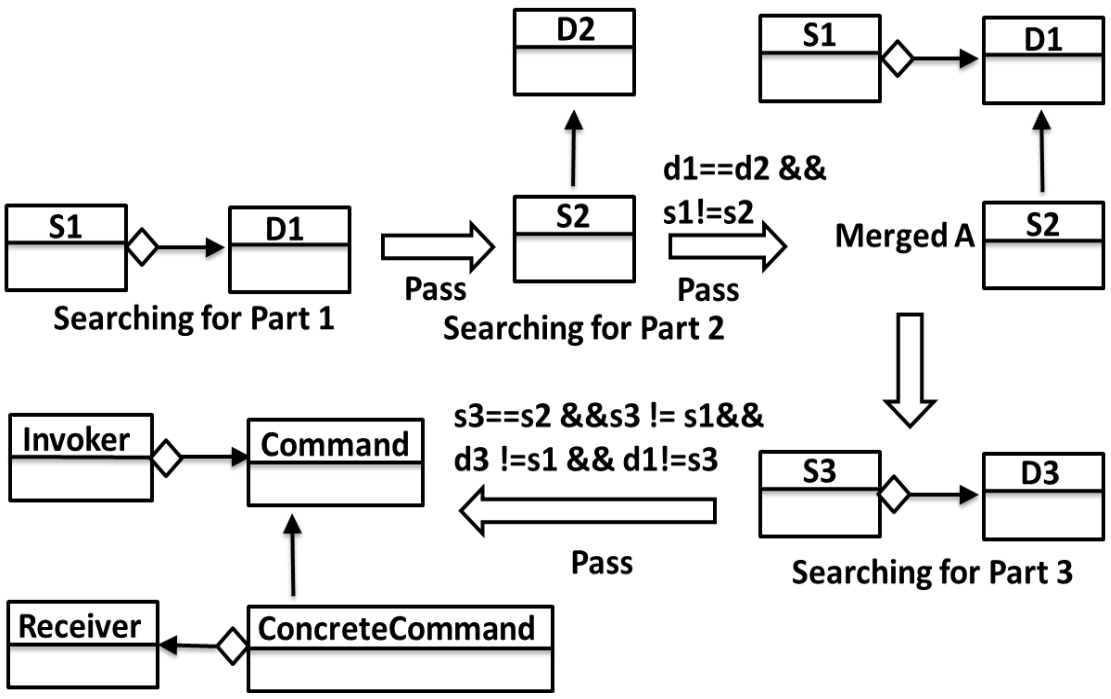
12. Proxy



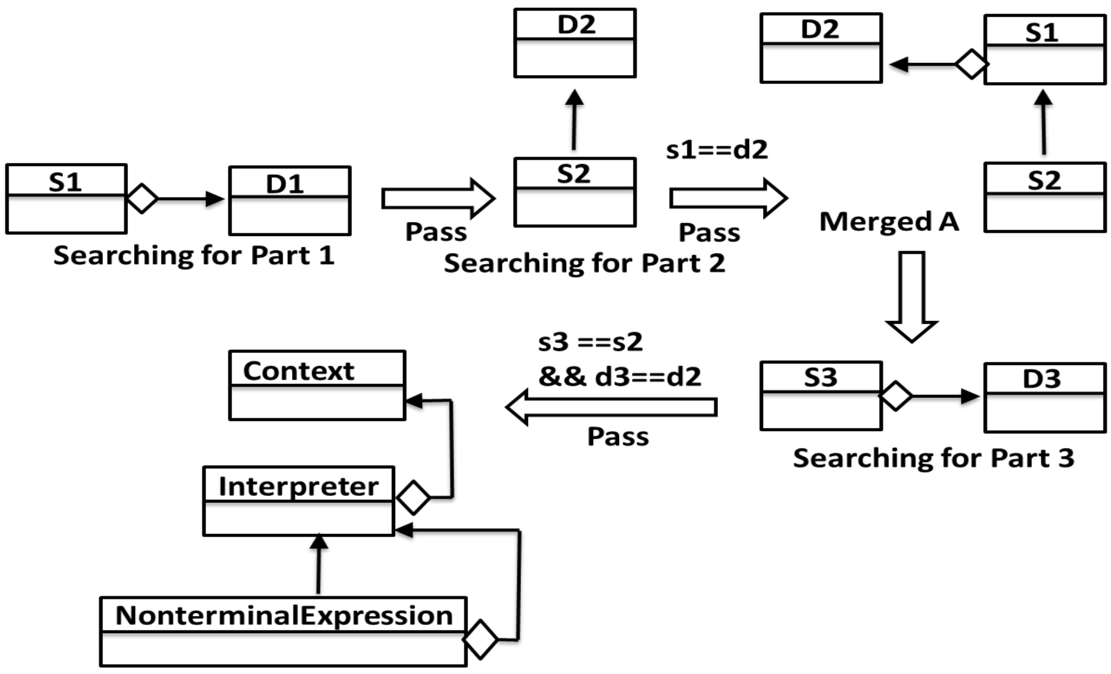
13. Chain of Responsibility



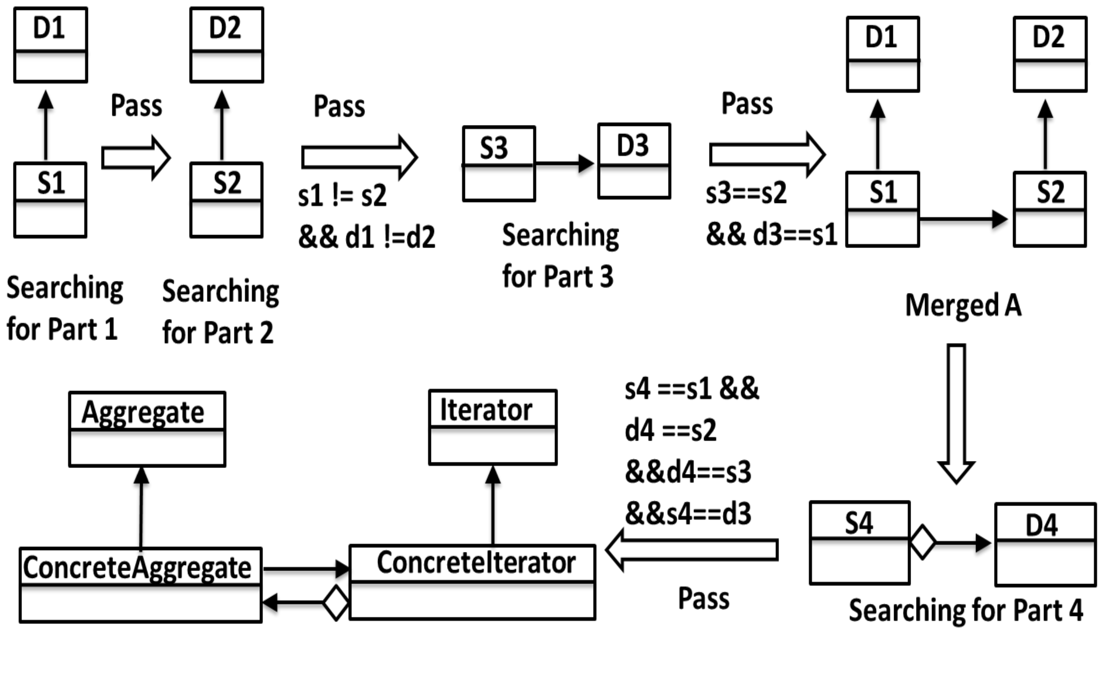
14. Command



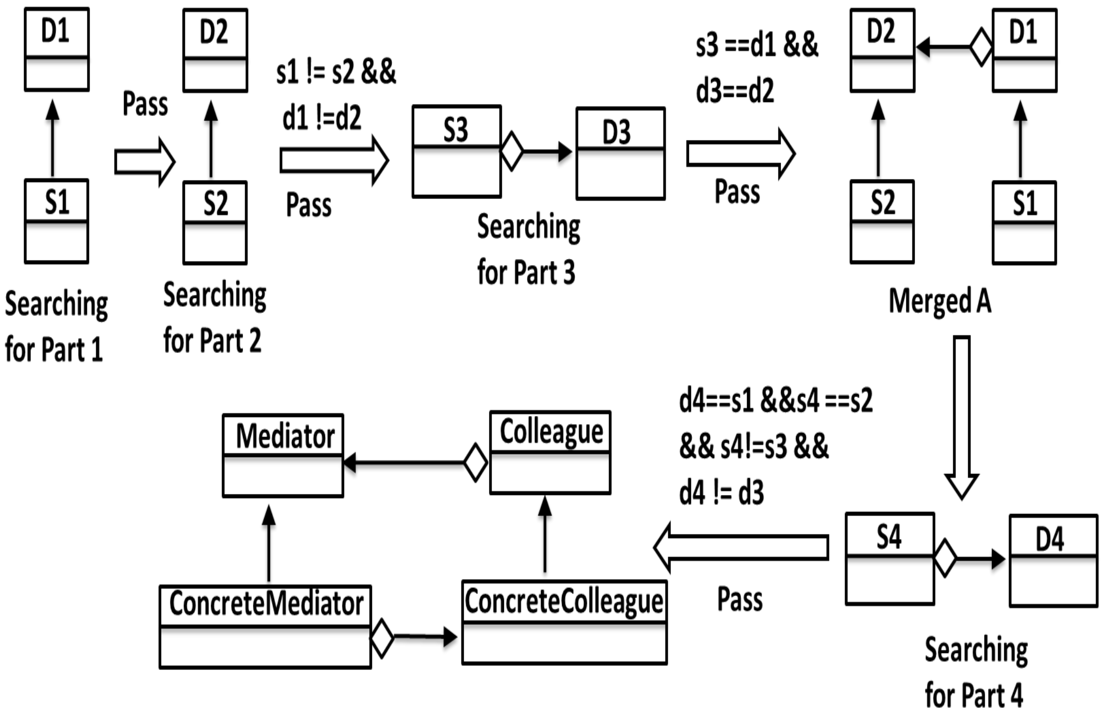
15. Interpreter



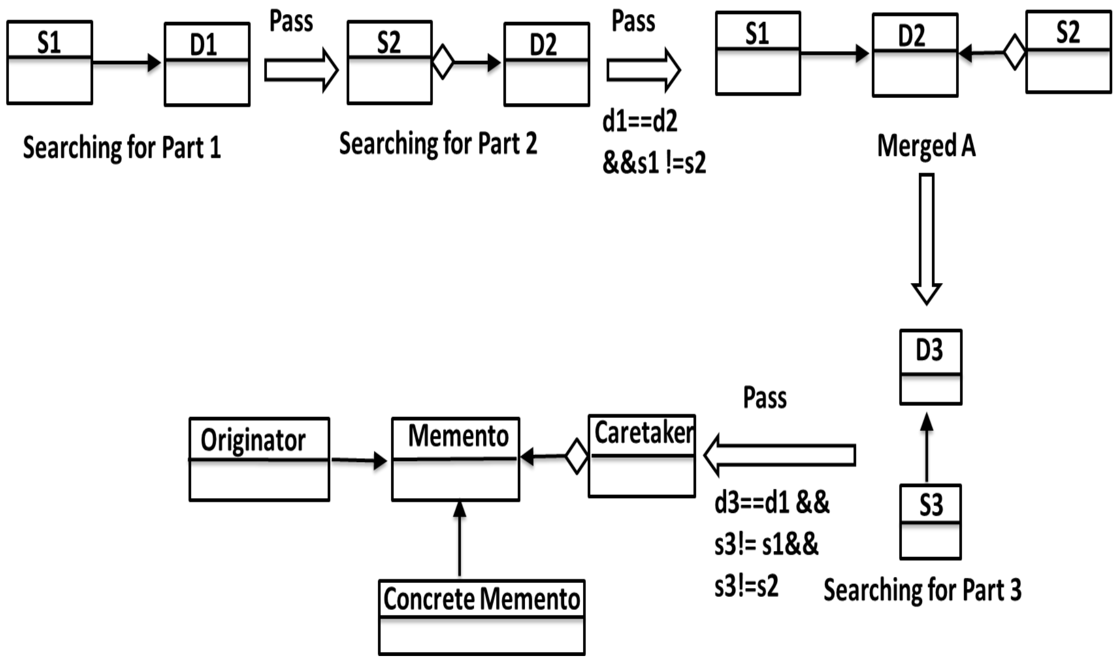
16. Iterator



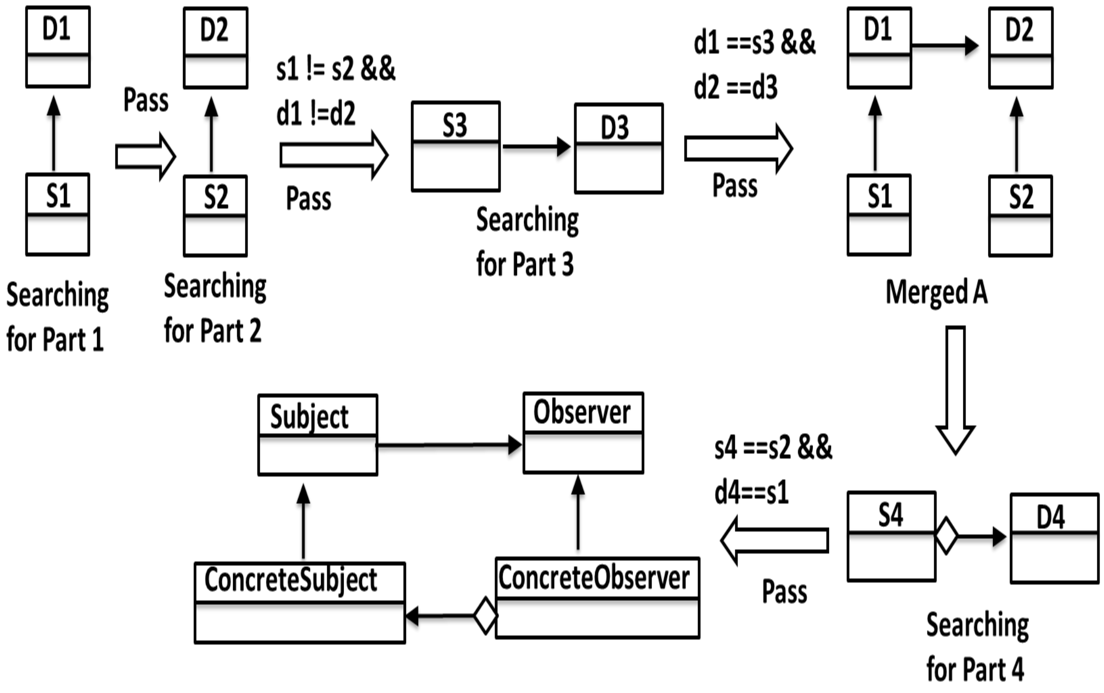
17. Mediator



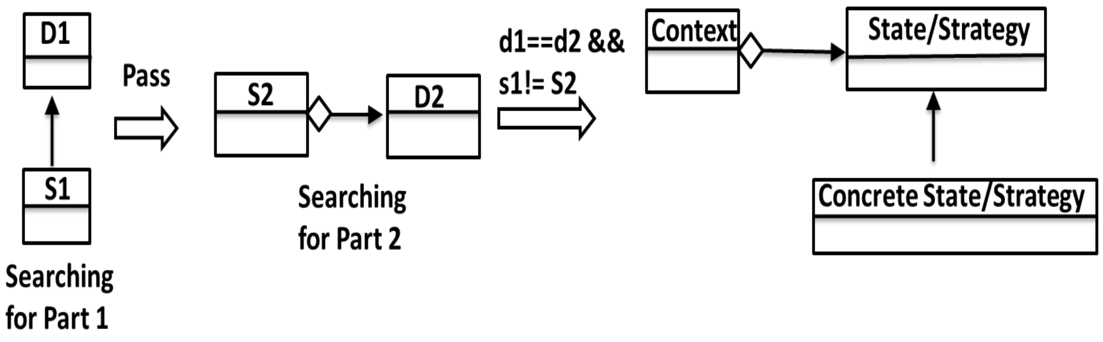
18. Memento



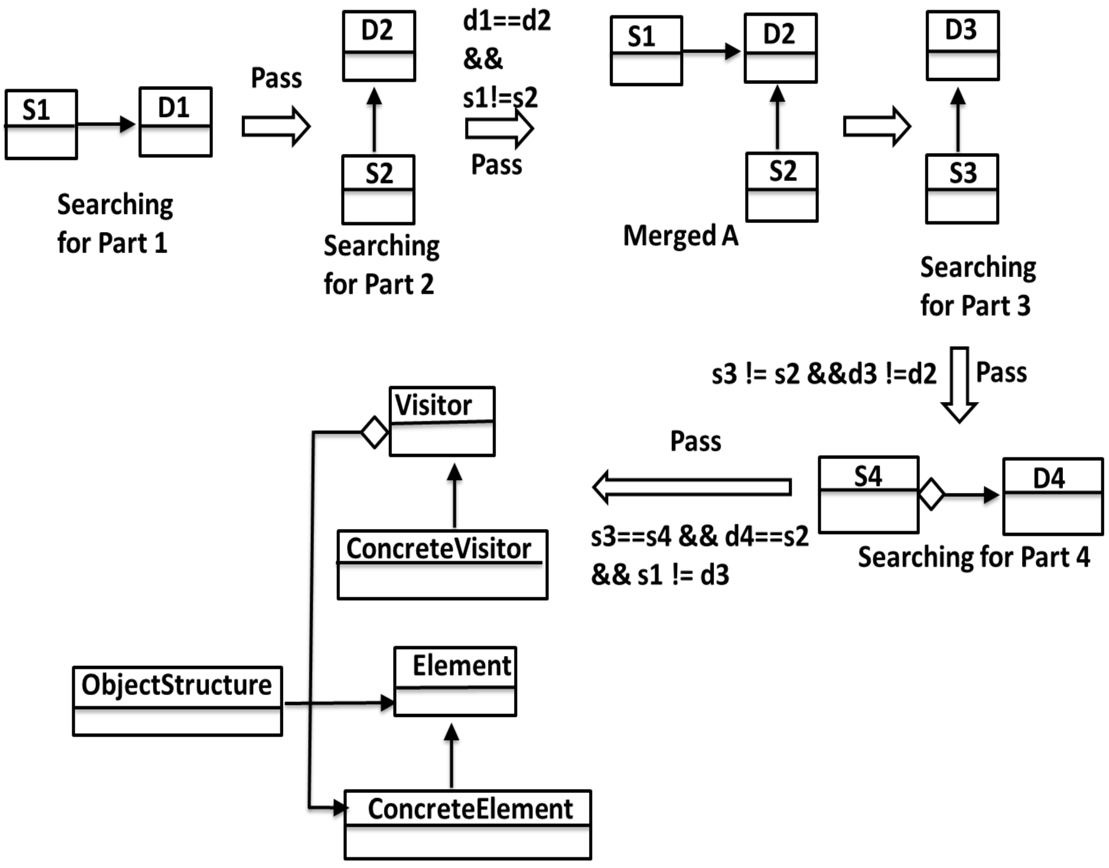
19. Observer



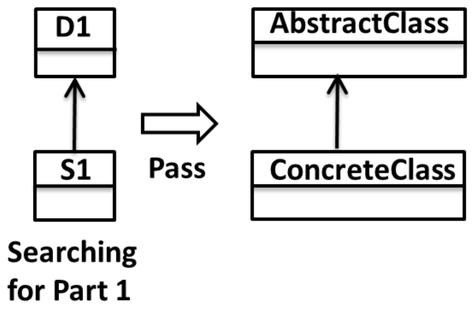
20. State/Strategy



21. Visitor



22. TemplateMethod



C Appendix

Pseudocode of the Structural Search Model (SSM)

Singleton_Instances

```
1. FOR each record in database {
2. IF (Connecting _Relationship == association) {
3. s1 = getSourceClass()       d1= getDestinationClass()
4. IF (s1== d1)   INSERT   s1   into Singleton
5. } // end IF
6. }// end FOR
```

Prototype_ instances

```
1. FOR each record in database { //1st FOR
2. IF (Connecting _Relationship == Realization) { // 1st IF
3. s1 =  getSourceClass   d1= getDestinationClass
4. FOR each record in database { //2nd FOR
5. IF (Connecting _Relationship == Dependency) { // 3rd IF
6.  s2 =  getSourceClass   d2= getDestinationClass
7.  IF(s1==s2 AND  d1==d2)  {  // 4th IF   ….prototype detected
8. INSERT    s1  into  Concrete _Prototype   and  INSERT    d1  into
   Prototype
9. END of 4th IF, 3rd IF, 2nd FOR, 1st IF, 1st FOR.
```

AbstractFactory_instances

```
1. FOR each record in database  { //1st FOR
2. IF (Connecting _Relationship == aggregation) { //1st IF
3. s1= getSourceClass     d1= getDestinationClass
4. FOR each record in database { //2ndFOR
5. IF (Connecting _Relationship == aggregation) { //2nd IF
6. s2= getSourceClass     d2= getDestinationClass
7. IF (s1 == s2  AND  d1 != d2) { //3rd IF
8. FOR each record in database  { //3rd  FOR
9. IF(Connecting _Relationship == Realization) { //5th IF
10. S3= getSourceClass     d3= getDestinationClass
11.tempConcreteFactory= s3
12.FOR each record in database  { //4th FOR
13.IF (Connecting _Relationship == Realization) {  // 6th IF
14.S4= getSourceClass     d4= getDestinationClass
15.IF (d4==d1 AND  d4!= d3) { //7rd IF
16.FOR each record in database {  //5th  FOR
17.IF (Connecting _Relationship == Realization) {   // 8th  IF
18.S5= getSourceClass    d5= getDestinationClass
19.IF (d2 == d5  AND d5!=d3)  { // 9th IF
20.tempProdcutA = s4    tempProductB= s5
21.FOR each record in database  {//6th FOR
22.IF (Connecting _Relationship == association)  {// 10th IF
23.S6= getSourceClass     d6= getDestinationClass
24.IF (d6==s4  OR d6==s5) {  //11th IF …. Abstract factory detected
25.INSERT   d1 into AbstractProductA
26.INSERT   d2 into AbstractProductB
27.INSERT   d3 into AbstractFactory
28.INSERT   s3 into ConcreteFactory
29.INSERT   s4 into ProductA
30.INSERT   s5 into ProductB
31.}//end  of 11th IF
32.}  // end  of 10th IF
33.}//end of 6th FOR
34.}// end  of 9th IF
35.}// end  of 8th  IF
36.}// end of 5th FOR
37.}// end of 7th IF
38.}// end of 6th IF
39.}//end of 4th  FOR
40.}//end of 5th IF
41.}// end of 3rd FOR
42.} //end of 3rd IF
43.}//end of 2nd IF
44.}// end of 2nd  FOR
45.} //end 1st  IF
46.}//end of 1st  FOR
```

Factory_instances

```
1.  FOR each record in database {   //1st FOR
2.  IF (Connecting _Relationship == inheritance) {     // 1st IF
3.  s1= getSourceClass     d1 = getDestinationClass
4.  FOR each record in database {  // 2nd FOR
5.  IF (Connecting _Relationship == Realization)  {//2nd IF
6.  s2= getSourceClass       d2 = getDestinationClass
7.  IF (s1!= s2 AND  d1 != d2) {  //3rd IF
8.  FOR each record in database {   // 3rd FOR
9.  IF (Connecting _Relationship == aggregation) {  // 4th IF
10. s3= getSourceClass  d3 = getDestinationClass
11. IF (s1==s3 AND  s2==d3) { // 5th IF  … factory detected
12. INSERT   d1 to Creator
13. INSERT   d2 to Product
14. INSERT   s1 to ConcreteCreator
15. INSERT   s2 to ConcreteProduct
16. }// end of 5th IF
17. } // end of 4th IF
18. }// end of 3rd  IF
19. }// end of 3rd FOR
20. } // end of 2rd IF
21. } //end of 2nd FOR
22. }// end of 1st IF
23. } // end of 1st FOR
```

Builder_instances

```
1.  FOR each record in database { // 1st FOR
2.  IF(Connecting _Relationship == association) { // 1st IF
3.  s1 = getSourceClass      d1= getDestinationClass
4.  FOR each record in database { //2nd FOR
5.  IF (Connecting _Relationship == Realization )  { // 2nd IF
6.  s2 = getSourceClass   d2= getDestinationClass
7.  IF (d1==d2) { // 3rd IF
8.  FOR each record in database { // 3rd FOR
9.  IF (Connecting _Relationship == association)  { // 4th IF
10. S3 = getSourceClass    D3= getDestinationClass
11. IF (s2==s3) {  // 5th IF …. Builder detected
12. INSERT   s1 into Director
13. INSERT   d1 into Builder
14. INSERT   s2 into ConcreteBuilder
15. INSERT   d3 into Product
16. }// end of 5th IF
17. }// end of 4th IF
18. }// end of 3rd FOR
19. }// end of 3rd IF
20. }// end of 2nd IF
21. }// end of 2nd FOR
22. }// end of 1st IF
23. }// end of 1st FOR
```

Adpater_instances

```
1.  FOR each record in database  {//1st FOR
2.    IF(Connecting _Relationship == Realization)   { //1st IF
3.      s1 = getSourceClass   d1 = getDestinationClass
4.      FOR each record in database {   // 2nd FOR
5.        IF (Connecting _Relationship == association) {  //2nd IF
6.          s2 = getSourceClass    d2 = getDestinationClass
7.          IF (s1==s2) { // 3rd IF    … adapter detected
8.            INSERT   d1 into Target
9.            INSERT   s1 into Adapter
10.           INSERT   d2 into Adaptee
11.         } // end of 3rd IF
12.       }// end of 2nd IF
13.     }// end of 2nd FOR
14.   }//  end of 1st IF
15. }// end of 1st  FOR
```

Bridge _instances

```
1.  FOR each record in database {  // 1st FOR
2.    IF (Connecting _Relationship == aggregation) {  //1st IF
3.      s1 = getSourceClass  d1= getDestinationClass
4.      FOR each record in database {  // 2nd FOR
5.        IF (Connecting _Relationship == Realization) { //  2nd IF
6.          s2 = getSourceClass;  d2= getDestinationClass;
7.          IF(s1==d2)  { // 3rd IF
8.            FOR each record in database { // 3rd FOR
9.              IF (Connecting _Relationship == Realization) { // 4th IF
10.               S3 = getSourceClass      D3= getDestinationClass
11.               IF (d3==d1) { //5th IF
12.                 INSERT   s3 into ConcreteImplementor
13.                 INSERT   d1 into Implementor
14.                 INSERT   d2 into Abstraction
15.                 INSERT   s2 into RefinedAbstraction
16.               }// end 5th IF
17.             }// end 4th IF
18.           } // end 3rd FOR
19.         }//end 3rd IF
20.       }// end 2nd IF
21.     }//end 2nd FOR
22.   }//end 1st IF
23. }// end 1st FOR
```

Composite_instances

```
1.  FOR record in database {    // 1st FOR
2.    IF (Connecting _Relationship == Realization) {  // 1st IF
3.      s1 = getSourceClass    d1= getDestinationClass
4.      FOR each record in database { //2nd FOR
5.        IF (Connecting _Relationship == association)  {  // 2nd IF
6.          s2 = getSourceClass   d2= getDestinationClass
7.          IF (s1== s2 AND d1==d2)  { // 3rd IF
8.            FOR each record in database { // 3rd FOR
9.              IF(Connecting _Relationship== Realization)  { //4th IF
10.               s3 = getSourceClass      d3= getDestinationClass
11.               IF (d3==d1)  AND  (s3 != s1) {  //5th IF
12.                 INSERT   s3 into Leaf
13.                 INSERT   d1 into Component
14.                 INSERT   s1 into Composite
15.               }//end of 5th IF
16.             } // end of 4th IF
17.           }// end of 3rd FOR
18.         }// end of 3rd IF
19.       }// end of 2nd IF
20.     }// end of 2nd FOR
21.   }// end of 1st IF
22. }// end of 1st FOR
```

Decorator_instances

```
1.  FOR each record in database {  // 1st FOR
2.    IF (Connecting _Relationship == aggregation) { // 1st IF
3.      s1 = getSourceClass     d1= getDestinationClass
4.      FOR (each record in database { // 2nd FOR
5.        IF (Connecting _Relationship == Realization )  { // 2nd IF
6.          s2 = getSourceClass   d2= getDestinationClass
7.          IF (s1==s2 AND d1==d2) { // 3rd IF
8.            FOR each record in database { // 3rd FOR
9.              IF (Connecting _Relationship == Realization)  {// 4th IF
10.               s3 = getSourceClass  d3= getDestinationClass
11.               IF (s1==d3) { // 5th IF
12.                 FOR each record in database { // 4rd FOR
13.                   IF (Connecting _Relationship == Realization) { // 6th IF
14.                     s4 = getSourceClass   d4= getDestinationClass
15.                     IF(s4!= s3 AND  s4 != s1 AND    d4 ==d1) { // 7th IF
16.                       INSERT   d1 into Component
17.                       INSERT   s1 into Decorator
18.                       INSERT   s3 into Concrete Decorator
19.                       INSERT   s4 into Concrete Component
20.}// end of 7th IF
21.}// end of 6th IF
22.}// end of 4th  FOR
23.} // end 5th IF
24.}// end of 4th IF
25.}// end of 3rd FOR
26.}// end of 3rd IF
27.}// end of 2nd IF
28.}// end of 2nd FOR
29.}// end of 1st IF
30.}// end of 1st FOR
```

Façade_instances

```
1.  FOR each record in database { // 1st FOR
2.  IF (Connecting _Relationship == aggregation)  {// 1st IF
3.  s1 = getSourceClass  d1= getDestinationClass
4.  FOR each record in database { // 2nd FOR
5.  IF (Connecting _Relationship == aggregation) { // 2nd IF
6.  s2 = getSourceClass   d2= getDestinationClass
7.  IF(d2 != d1)  { // 3rd IF , not with itself
8.  FOR each record in database  { //3rd FOR
9.  IF (Connecting _Relationship == aggregation) { //4th IF
10. s3 = getSourceClass   d3= getDestinationClass
11. IF(d3 !=d1) { //5th IF
12. IF ( s1==s2 AND  s1==s3){ // 6th IF
13. INSERT   s1 into Façade
14. INSERT   d1 into Subsystem1
15. INSERT   d2 into Subsystem2
16. INSERT   d3 Subsystem3
17. } // end of 6th IF
18. }// end of 5th IF
19. }// end of 4th IF
20. }// end of 3rd FOR
21. }// end of 3rd IF
22. }// end of 2nd IF
23. }// end of 2nd FOR
24. }// end of 1st IF
25. }// end of 1st FOR
```

Flyweight_instances

```
1.  FOR each record in database {  //1st FOR
2.  IF (Connecting _Relationship == association) { // 1st IF
3.  s1 = getSourceClass       d1= getDestinationClass
4.  FOR each record in database { // 2nd FOR
5.  IF (Connecting _Relationship== inheritance ) //2nd IF
6.  s2 = getSourceClass  d2= getDestinationClass
7.  IF ( d2==d1 AND  s2 != s1) { //3rd IF
8.  FOR each record in database { //3rd FOR
9.  IF (Connecting _Relationship == inheritance ) { //4th IF
10. s3 = getSourceClass    d3= getDestinationClass
11. IF (d3==d1 AND  s2 !=s3)  { // 5th IF
12. INSERT   s1 into FlyweightFactory
13. INSERT   d1 into Flyweight
14. INSERT   s2 into UnsharedConcreteFlyweight
15. INSERT   s3 into ConcreteFlyweight
16. } // end of 5th IF
17. }// end of 4th IF
18. }// end of 3rd FOR
19. }// end of 3rd IF
20. }// end of 2nd IF
21. }// end of 2nd FOR
22. }// end of 1st IF
23. }// end of 1st FOR
```

Proxy_instances

```
1.  FOR each record in database {// 1st FOR
2.  IF (Connecting _Relationship == Realization) {// 1st IF
3.  s1 = getSourceClass       d1= getDestinationClass
4.  FOR each record in database {// 2nd FOR
5.  IF (Connecting _Relationship == Realization){ //2nd IF
6.  s2 = getSourceClass      d2= getDestinationClass
7.  IF( s1 != s2 AND  d1==d2) {//3rd IF
8.  FOR each record in database  {// 3rd FOR
9.  IF (Connecting _Relationship= = association)  {//4th IF
10. s3 = getSourceClass    d3= getDestinationClass
11. IF(s3==s1 AND  d3==s2)  {//5th IF
12. INSERT   d1 into Subject
13. INSERT   s1 into Proxy
14. INSERT   s2 into RealSubject
15. } // end of 5th IF
16. }// end of 4th IF
17. }// end of 3rd FOR
18. }// end of 3rd IF
19. }// end of 2nd IF
20. }// end of 2nd FOR
21. }// end of 1st IF
22. }// end of 1st FOR
```

ChainofResponsibility_instances

```
1.  FOR each record in database {  // 1st FOR
2.    IF ( Connecting _Relationship == aggregation) { // 1st IF
3.      s1 = getSourceClass     d1= getDestinationClass
4.      IF (s1==d1) {// 2nd IF , aggregation with itself
5.        FOR each record in database { // 2nd FOR
6.          IF (Connecting _Relationship == Realization) {  //3rd IF
7.            s2 = getSourceClass      d2= getDestinationClass
8.            IF(d2==s1) { //4th IF
9.              INSERT   s1 into Handler
10.             INSERT   s2 into ConcreteHandler
11.           }// end of 4th IF
12.         }// end of 3rd IF
13.       }// end of 2nd FOR
14.     }// end of 2nd IF
15.   }// end of 1st IF
16. }// end of 1st FOR
```

Command_instances

```
1.  FOR each record in database {// 1st FOR
2.    IF (Connecting _Relationship == aggregation){ // 1st IF
3.      s1 = getSourceClass    d1= getDestinationClass
4.      FOR each record in database { // 2nd FOR
5.        IF (Connecting _Relationship == INHERITANCE) { //2nd  IF
6.          s2 = getSourceClass     d2= getDestinationClass
7.          IF(d1==d2) { //3rd  IF
8.            FOR each record in database {// 3rd FOR
9.              IF (Connecting _Relationship == aggregation) {//4th IF
10.               s3 = getSourceClass    d3= getDestinationClass
11.               IF (s3==s2 AND  s3 != s1 AND  d3 !=s1){ //5th IF
12.                 INSERT   s1 into Invoker
13.                 INSERT   d1 into Command
14.                 INSERT   s2 into ConcreteCommand
15.                 INSERT   d3 into Receiver
16.               }// end of 5th IF
17.             }// end of 4th IF
18.           }// end of 3rd FOR
19.         }// end of 3rd IF
20.       }// end of 2nd  IF
21.     }// end of 2nd FOR
22.   }// end of 1st IF
23. }// end of 1st FOR
```

Interpreter_instances

```
1.  FOR each record in database {// 1st FOR
2.    IF (Connecting _Relationship == aggregation { // 1st IF
3.      s1 = getSourceClass    d1= getDestinationClass
4.      FOR each record in database { //2nd  FOR
5.        IF(Connecting _Relationship == Realization) { // 2nd IF
6.          s2 = getSourceClass     d2= getDestinationClass
7.          IF(s1==d2) { //3rd IF
8.            FOR each record in database {// 3rd FOR
9.              IF(Connecting _Relationship == aggregation) { //4th IF
10.               s3 = getSourceClass;
11.               d3= getDestinationClass;
12.               IF (s3 ==s2 AND  d3==d2) { //5th IF
13.                 INSERT   s1 into AbstractExpression
14.                 INSERT   d1 Context
15.                 INSERT   s2 into NonterminalExpression
16.               }// end of 5th IF
17.             }// end of 4th IF
18.           }// end of 3rd FOR
19.         }// end of 3rd IF
20.       }// end of 2nd IF
21.     }// end of 2nd FOR
22.   }// end of 1st IF
23. }// end of 1st FOR
```

Iterator_instances

```
1.  FOR each record in database // 1st FOR {
2.    IF (Connecting _Relationship == Realization ) // 1st IF  {
3.      s1 = getSourceClass     d1= getDestinationClass
4.      FOR each record in database {// 2nd FOR
5.        IF (Connecting _Relationship == Realization) {// 2nd IF
6.          s2 = getSourceClass  d2= getDestinationClass
7.          IF (s1 != s2 AND  d1 !=d2) {//3rd IF
8.            FOR each record in database {// 3rd FOR
9.              IF (Connecting _Relationship ==  association) {//4th IF
10.               s3 = getSourceClass      d3= getDestinationClass
11.               IF (s3==s2 AND d3 ==s1) { // 5th IF
12.                 FOR each record in database {// 4th FOR
13.                   IF(Connecting _Relationship == aggregation ) { //6th IF
14.                     s4 = getSourceClass      d4= getDestinationClass
15.                     IF (s4 ==s1 AND  d4 ==s2  AND  d4==s3 AND  s4==d3){ //7th IF
16.                       INSERT   d1 into Iterator
17.                       INSERT   d2 into Aggregate
18.                       INSERT   s1 into ConcreteIterator
19. INSERT   s2 into ConcreteAggregate
20. }// end of 7th IF
21. }// end of 6th IF
22. }// end of 4th FOR
23. }// end of 5th IF
24. }// end of 4th IF
25. }// end of 3rd FOR
26. }// end of 3rd IF
27. }// end of 2nd IF
28. }// end of 2nd FOR
29. }// end of 1st IF
30. }// end of 1st FOR
```

Mediator_instances

```
1.  FOR each record in data base {// 1st FOR
2.  IF (Connecting _Relationship == inheritance) { // 1st IF
3.  s1 = getSourceClass    d1= getDestinationClass
4.  FOR each record in data base {// 2st FOR
5.  IF (Connecting _Relationship == inheritance) { // 2st IF
6.  s2 = getSourceClass     d2= getDestinationClass
7.  (IF s1!=s2 AND  d1 != d2) {// 3rd IF
8.  FOR each record in database { // 3rd FOR
9.  IF (Connecting _Relationship == aggregation) { // 4th IF
10. s3 = getSourceClass  d3= getDestinationClass
11. IF (s3 ==d1 AND   d3==d2) { // 5th IF
12. FOR each record in data base { //4th FOR
13. IF (Connecting _Relationship == aggregation) { //6th IF
14. s4 = getSourceClass  d4= getDestinationClass
15. IF ( d4==s1 AND  s4 ==s2 s AND 4!=s3 AND  d4 != d3) { //7th IF
16. INSERT   d1 into Colleague
17. INSERT    s1 into ConcreteColleague
18. INSERT    s2 into ConcreteMediator
19. INSERT    d2 into Mediator
20. }// end of 7th IF
21. }// end of 6th IF
22. }// end of 4th FOR
23. }// end of 5th IF
24. }// end of 4th IF
25. }// end of 3rd FOR
26. }// end of 3rd IF
27. }// end of 2nd IF
28. }// end of 2nd FOR
29. }// end of 1st IF
30. }// end of 1st FOR
```

Memento_instances

```
1.  FOR each record in database {// 1st FOR
2.  IF(Connecting _Relationship == ASSOCIATION) {// 1st IF
3.  s1 = getSourceClass  d1= getDestinationClass
4.  FOR each record in database {// 2nd FOR
5.  IF(Connecting _Relationship == aggregation) {// 2nd  IF
6.  s2 = getSourceClass  d2= getDestinationClass
7.  IF(d1==d2 AND  s1 !=s2) {// 3rd IF
8.  FOR each record in database {// 3rd FOR
9.  IF(Connecting _Relationship == inheritance){ //4th if
10. s3 = getSourceClass  d3= getDestinationClass
11. if(d3==d1 && s3!= s1 && s3!=s2){// 5th if
12. INSERT    d1 into Memento
13. INSERT   s1 into Originator
14. INSERT   s2 into Caretaker
15. INSERT   s3 into ConcreteMemento
16. }// end of 5th if
17. }// end of 4th if
18. }//end 3rd for
19. } // end of 3rd IF
20. }// end of 2nd IF
21. }// end of 2nd FOR
22. }// end of 1st IF
23. }// end of 1st FOR
```

Observer_instances

```
1.  FOR each record in database {// 1st FOR
2.  IF (Connecting _Relationship == Realization ){ //1st IF
3.  s1 = getSourceClass  d1= getDestinationClass
4.  FOR each record in database {// 2nd  FOR
5.  IF (Connecting _Relationship == Realization ){ //2nd IF
6.  s2 = getSourceClass  d2= getDestinationClass
7.  IF (s1!= s2 AND  d1 != d2){//3rd IF
8.  FOR each record in database {// 3rd FOR
9.  IF (Connecting _Relationship == association ){//4th IF
10. s3 = getSourceClass  d3= getDestinationClass
11. IF (d1 ==s3 AND d2 ==d3) {//5th IF
12. FOR each record in database {//4th FOR
13. IF(Connecting _Relationship == aggregation ){ //6th IF
14. s4 = getSourceClass     d4= getDestinationClass
15. IF ( s4 ==s2 AND  d4==s1) {//7th IF
16. INSERT   d1 into Subject
17. INSERT   s1 ConcreteSubject
18. INSERT   d2 into Observer
19. INSERT   s2 into ConcreteObserver
20. }// end of 7th IF
21. }// end of 6th IF
22. }// end of 4th FOR
23. }// end of 5th IF
24. }// end of 4th IF
25. }// end of 3rd FOR
26. }// end of 3rd IF
27. }// end of 2nd IF
28. }// end of 2nd FOR
29. }// end of 1st IF
30. }// end of 1st FOR
```

State_Strategy_instances

```
1.  FOR each record in data base {//1st FOR
2.  IF (Connecting _Relationship == aggregation ) { //1st IF
3.  s1 = getSourceClass    d1= getDestinationClass
4.  FOR each record database { // 2nd  FOR
5.  IF (Connecting _Relationship == Realization){//2nd IF
6.  s2 = getSourceClass    d2= getDestinationClass
7.  IF(d1==d2 AND  s1!= S2) { //3rd IF
8.  INSERT   S1 into Context
9.  INSERT   d1 into State
10. INSERT   s2 into ConcreteState
11. }// end of 3rd IF
12. }// end of 2nd IF
13. }// end of 2nd FOR
14. }// end of 1st IF
15. }// end of 1st FOR
```

Template_instances

```
1.  FOR each record in database {// 1st FOR
2.  IF (Connecting _Relationship == inheritance) {// 1st IF
3.  s1 = getSourceClass       d1= getDestinationClass
4.  IF( s1 != d1){ // 2nd  IF
5.  INSERT   s1 into ConcreteClass
6.  INSERT   d1 into AbstractClass
7.  }// end of 2nd  IF
8.  }// end of 1st  IF
9.  }// end of 1st  FOR
```

Visitor_instances

```
1.  FOR each record in database {// 1st FOR
2.  IF (Connecting _Relationship == association ) {//1st IF
3.  s1 = getSourceClass       d1= getDestinationClass
4.  FOR each record in database {    //2nd for
5.  IF (Connecting _Relationship == Realization) {// 2nd IF
6.  s2= getSourceClass      d2= getDestinationClass
7.  IF ( d1==d2 AND  s1!=s2) { //3rd IF
8.  FOR each record in database {// 3rd FOR
9.  IF (Connecting _Relationship == Realization ) {// 4th IF
10. s3= getSourceClass    d3= getDestinationClass
11. IF(s3 != s2 AND  d3 !=d2){// 5th IF
12. FOR each record in database {//4th FOR
13. IF (Connecting _Relationship == aggregation ) {// 6th IF
14. s4= getSourceClass        d4= getDestinationClass
15. IF(s3==s4 AND d4==s2 AND  s1 != d3){//7th IF
16. INSERT   s1 into ObjectStructure
17. INSERT   d1 into Element
18. INSERT    s2 into ConcreteElement
19. INSERT   d3 into Visitor
20. INSERT   s3 into ConcreteVisitor
21. }// end of 7th IF
22. }// end of 6th IF
23. }// end of 4th FOR
24. }// end of 5th IF
25. }// end of 4th IF
26. }// end of 3rd FOR
27. }// end of 3rd IF
28. }// end of 2nd IF
29. }// end of 2nd FOR
30. }// end of 1st IF
31. }// end of 1st FOR
```

D Appendix
List of Publications

[Chapter Two]

1. Al-Obeidallah, M.G., Petridis, M. and Kapetanakis, S.: 'A Survey on Design Pattern Detection Approaches', International Journal of Software Engineering, 7(3), pp. 41–59. (2016).

[Chapters Three and Four]

2. Al-Obeidallah, M.G., Petridis, M. and Kapetanakis, S.: MLDA: A Multiple Levels Detection Approach for Design Patterns Recovery, in the proceedings of 2017 International Conference On Computing and Data Analysis (ICCDA 2017), May 19-23, Florida, United States, 2017, pp. 33-40.

[Chapters Five and Six]

3. Al-Obeidallah M.G., Petridis M., Kapetanakis S. (2018). A Structural Rule-Based Approach for Design Patterns Recovery. In: Lee R. (eds) Software Engineering Research, Management and Applications. SERA 2017. Studies in Computational Intelligence, vol 722. Springer, Cham.

Accepted Papers

[Chapters Three until Six] (The complete model with all datasets)

4. Al-Obeidallah M.G., Petridis M., Kapetanakis S. A Multiple Phases Approach for Design Patterns Recovery Based on Structural and Method Signature Features. Accepted for publication in the International Journal of Software Innovation (IJSI), vol. 6, issue 3, December 2017.

IJSI is indexed by **Thomson Reuters**, Scopus, ACM, Web of science, INSPEC, etc.

Awards

1. Best Student Paper Award (SERA 2017)

"A Structural Rule-Based Approach for Design Patterns Recovery", which has been published in the 15th ACIS International Conference on Software Engineering Research, Management and Applications (SERA 2017) http://www.acisinternational.org/sera2017/, has won the best student paper award in the conference. This paper was selected out of 80 submissions.