A GRAPH MINING APPROACH FOR DETECTING DESIGN PATTERNS IN OBJECT-ORIENTED DESIGN MODELS

NESNE-TABANLI YAZILIM MODELLERİNDE BENZER TASARIM YAPILARINI TESPİT EDEN ÇİZGE MADENCİLİĞİ YÖNTEMİ

MURAT ORUÇ

ASST. PROF. DR. FUAT AKAL
Supervisor

Submitted to Graduate School of Science and Engineering of Hacettepe University
as a Partial Fulfillment to the Requirements
for the Award of the Degree of Master of Science
in Computer Engineering

2016

This work named "**A Graph Mining Approach For Detecting Design Patterns in Object-Oriented Design Models**" by **MURAT ORUÇ** has been approved as a thesis fort he Degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING** by the below mentioned Examining Committee Members.

Prof. Dr. Hayri SEVER
Head

Asst. Prof. Dr. Fuat AKAL
Supervisor

Prof. Dr. Onur DEMİRÖRS
Member

Assoc. Prof. Dr. Lale ÖZKAHYA
Member

Asst. Prof. Dr. Ayça TARHAN
Member

This thesis has been approved as a thesis fort he Degree of **MASTER OF SCIENCE IN COMPUTER ENGINEERING** by Board of Directors of the Institute for Graduate School of Science and Engineering.

Prof. Dr. Salih Bülent ALTEN
Director of the Institute of
Graduate School of Science and Engineering

ETHICS

In this thesis study, prepared in accordance with the spelling rules of Institute of Graduate Studies in Science of Hacettepe University,

I declare that

- all the information and documents have been obtained in the base of the academic rules
- all audio-visual and written information and results have been presented according to the rules of scientific ethics
- in case of using others Works, related studies have been cited in accordance with the scientific standards
- all cited studies have been fully referenced
- I did not do any distortion in the data set
- and any part of this thesis has not been presented as another thesis study at this or any other university.

12/04/2016

MURAT ORUÇ

ABSTRACT

A GRAPH MINING APPROACH FOR DETECTING DESIGN PATTERNS IN OBJECT-ORIENTED DESIGN MODELS

Murat Oruç

Graduate School, Computer Engineering

Supervisor: Asst. Prof. Dr. Fuat Akal

April 2016, 91 Pages

Object-oriented design patterns are frequently used in real-world applications. As design patterns are the common solutions for recurring problems which software developers confronted with, they help developers to implement the design easily. Design patterns also demonstrate the code reusability and strengthen the quality of the source code. Therefore, detection of design patterns is essential for comprehension of the intent and design of a software project. This thesis presents a graph-mining approach for detecting design patterns. The approach of detection process is based on searching sub-graphs of input design patterns in the space of model graph of the source code by isomorphic sub-graph search method. Within the scope of this thesis, 'DesPaD' (Design Pattern Detector) tool is developed for detecting design patterns. To implement the isomorphic search, open-source sub-graph mining tool, Subdue is used. The examples of 23 GoF design patterns in the book of "Applied Java Patterns" are detected and some promising results in JUnit 3.8, JUnit 4.1 and Java AWT open-source packages are obtained.

Keywords: sub-graph mining, design patterns, object-oriented, software architecture.

ÖZET

NESNE-TABANLI YAZILIM MODELLERİNDE BENZER TASARIM YAPILARINI TESPİT EDEN ÇİZGE MADENCİLİĞİ YÖNTEMİ

Murat Oruç

Yüksek Lisans, Bilgisayar Mühendisliği

Tez Danışmanı: Yrd. Doç. Dr. Fuat Akal

Nisan 2016, 91 sayfa

Nesne-tabanlı tasarım kalıpları gerçek-dünya uygulamalarında sıklıkla kullanılmaktadır. Tasarım kalıpları yazılım geliştiricilerin karşılaştıkları tekrar eden problemlere ortak çözümler olduğu için, yazılım geliştiricilere tasarımın kolayca uygulanmasını sağlar. Tasarım kalıpları ayrıca kaynak kodun kalitesi ve kodun yeniden kullanılabilirliğini gösterir. Bu yüzden, tasarım kalıplarını tespit etme yazılım projesinin tasarımını ve niyetini anlamada önem arz etmektedir. Bu tez, tasarım örüntülerini tespit eden bir çizge madenciliği yaklaşımını sunmaktadır. Tespit işlemi yaklaşımı, kaynak kodun model çizgesinin uzayında, izomorfik alt-çizge arama metodu aracılığıyla tasarım desenleri girdilerinin alt-çizgelerini aramaya dayalıdır. Tez kapsamında, tasarım örüntülerini tespit etmek için 'DesPaD' (Tasarım Deseni Detektörü) aracı geliştirilmiştir. İzomorfik aramayı uygulamak için açık-kaynak kodlu alt-çizge madenciliği aracı olan Subdue adlı referans kullanılmıştır. "Applied Java Patterns" kitabıyla beraber gelen 23 GoF tasarım deseni örnekleri tespit edilmiş, ayrıca yapılan deneylerde JUnit 3.8, JUnit 4.1 ve Java AWT açık-kaynak yazılımlarında bazı cesaretlendirici sonuçlar elde edilmiştir.

Anahtar Kelimeler: Alt-çizge madenciliği, tasarım örüntüleri, tasarım deseni, tasarım kalıbı, nesne-tabanlı, yazılım mimarisi.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Asst. Prof. Dr. Fuat Akal and my co-advisor Prof. Dr. Hayri Sever of the Computer Engineering Department at Hacettepe University. The door to Asst. Prof. Dr. Akal and Prof. Dr. Sever offices were always open whenever I ran into a trouble spot or had a question about my research or writing. They consistently allowed this paper to be my own work, but steered me in the right the direction whenever they thought I needed it.

I would never have been able to finish my thesis without the support of my wife and my little son. My eternal love belongs to them. This accomplishment would not have been possible without them. Thank you.

TABLE OF CONTENTS

FIGURES

Page

TABLES

Page

1. INTRODUCTION

1.1. Statement of the Problem

Using design patterns has become increasingly well known in software development since 1990s in object-oriented programming. The book, "*Design Patterns: Elements of Reusable Object-Oriented Software* [4] by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (known in the industry as the Gang of Four, or GoF)" is generally believed with the increasing concern in design patterns for use in object-oriented models. The book consists of twenty-three patterns. Each of these patterns has a fix solving for a reemerging problem encountered in object-oriented design. This thesis focused on detecting 23 GoF patterns.

Object-oriented principles and reusable design patterns are frequently used in software projects. Detecting such patterns would benefit in many ways. First, due to lack of and poor documentation, it would typically take a long time for a developer to comprehend the design of the entire source code. As developers of a software project may change during the project life cycle, getting insights of the source code for the new developers is usually a repeating and challenging process. Therefore, it is crucial to have a tool for revealing the intent and design of a software project. As design patterns are used for settling up a common recurring design problem in a specific context in terms of reusable object-oriented design, they are important for comprehending software architecture and evaluating its nature and characteristics.

Second, Booch stated in his "Draw Me a Picture" [30] article that the patterns in software projects are essential to comprehend its architecture and to check its quality. He advised programmers that they should draw a picture that highlights a pattern scattered across their code to see the architecture and design of that software project. So, detection of the structures like design patterns and visualization of them will present the opportunity to develop and see the hidden architecture for the software team. By the help of this, they easily may understand the partial intent and see the big picture of that software.

Furthermore, keeping up maintenance tasks on a software project takes more than two-third of the total cost, where comprehension activities constitute considerable amount [1, 2]. Detection of design patterns may be the auxiliary tool for comprehension of the project in maintenance tasks.

Consequently, a design pattern detection tool for an object-oriented software project is essential because, by using such a tool, intent, design and general view of a software project can be extracted easily.

1.2. Definitions

This section gives a list of definitions that are related to this thesis. These definitions are made to guide the reader and build a common language. Terms are generally related with the issues of graph theory and software engineering.

- **Object-Oriented Programming:** It is a programming pattern based on the concept of objects that contains attributes and methods.
- **Some basic definitions in object-oriented programming:**
  - **Association:** It is a relationship indicating a semantic connection between two classes [38].
  - **Inheritance:** It includes a relationship among classes. The structure or behavior identified in one class is shared by one or more other classes [38].
  - **Abstract / Concrete Classes:** A class may be abstract or concrete. Concrete class is a class which has a total implementation. Therefore, instances may be produced from concrete class. However, an abstract class is a class whose concrete subclasses may add to its structure and behavior [39].
  - **Interfaces:** While a list of operations of a class or a component is determined in an interface, operations’ behaviors are defined within the classes that implement that interface. [39].
- **Software Design Pattern:** It is a reusable code design within a given context in the layout of a software project that helps to solve a reemerging problem. [4].

- **Graph:** Formally, its definition is given as “A formation by vertices and edges connecting the vertices” [33]. G is a graph defined by G = (V, E). V is a set of vertices or nodes and E is a set of edges or arcs [19].

- **Directed Graph (Digraph):** It is defined as “A graph, or cluster of vertices connected by edges, where the edges have a direction related with them”. When G is a graph defined by G = (V, E), V is a set of vertices or nodes, E is a set of directed edges or arcs [19].

- **Sub-graph:** It is described as “A graph is a sub-graph of G, defined as $G_s \subseteq G$, if the vertices and edges of G_s embodies a subset of the vertices and edges of G ($V_s \subseteq V$ and $E_s \subseteq E$)” [11].

- **Isomorphic sub-graph:** It is defined as “The two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic when labeling the vertices of G_1 bijectively with the elements of V_2 gives G_2 and multiplicity of edges are maintained” [11].

1.3. Research Contribution

Detecting design patterns from a software project attracted attention after object-oriented design principles were established and design patterns like GRASP [3] and GoF [4] were described. Within this context, capturing static and dynamic aspects of the software by using reverse-engineering methods [5, 6], defining patterns based on software metrics and their roles [7, 8], identification of micro-architectures similar to design patterns [9, 10] and some graph-based approaches [11, 12, 13, 14, 15] are published in the literature. The approach of this thesis is to detect design patterns, on the other hand, is based on building a high-level model graph of a given software project, representing design patterns as graphs and implementing sub-graph mining search using open-source tool, Subdue [16, 17]. High-level understanding of a project is targeted by extracting and visualizing design patterns used in it, which will help developers or architects of the project to figure it out conveniently.

For the experimental of this work a fully automated tool, DesPaD (Design Pattern Detector) for detecting design patterns were developed. Several experiments by using the demo source code came with the Applied Java Patterns text book [23] and also on some open software projects namely JUnit 3.8, JUnit 4.1 and Java AWT projects were conducted. Results of the experiments are promising.

1.4. Thesis Outline

This thesis consists of six sections. In the first section an introduction to design patterns is explained. The motivation behind detecting design patterns in a software project is also described in this section. This section also explains the contribution of thesis on the field of design pattern detection and the definitions used within the thesis.

Section 2 introduces the background information and related work on design pattern detection. Some different approaches implemented for detecting design patterns in the literature and some of the tools for detecting design patterns are presented. The differences between related studies and the approach of this thesis are also stated in this chapter.

Section 3 demonstrates the architecture of detecting design patterns in this work. The methodology is explained in three steps, “Model Graph Creation”, “Design Pattern Template Creation” and “Detecting Design Patterns”.

Section 4 presents the Design Pattern Detector (DesPaD) tool in detail. In this part the architecture of the DesPaD tool and the auxiliary-tools it uses are described.

Section 5 points out the evaluations, which are applied to test the correctness, functionality and performance of the DesPaD tool. Design pattern examples in the “*Applied Java Patterns”* book written by “*Stephen A. Stelting”* and “*Olav Maassen”* are benefited to demonstrate that GoF design patterns can be detected in a correct way in DesPaD. The DesPaD tool is also compared with other design pattern detection tools in terms of functionality.

Section 6 concludes the work and summarizes the obtained results. An overview of possible future work is also explained in this section.

2. BACKGROUND

2.1. Literature Overview

There are graph-based design pattern detection studies submitted in the literature [12, 13, 14, 15].

A template matching method is implemented to detect design patterns in a given source code in [12]. They determine some features of design patterns to create templates and, then look them up in a given source code. As compared to this thesis, their approach does not go deep into pattern specifics as much as this study does. In this thesis, a model graph of the given source code with twelve relation types, which is more specific than design features implemented in [12] is built. For instance, while they have a single design feature to cover generalization pattern, in this thesis three specific relations, which are extends, implements and overrides to cover the same feature are used. Building a detailed model graph is more prone to preventing false-positives while detecting patterns. Calculating the similarity scores of each vertex in matrices representing the features of patterns is used for detecting design patterns in [13]. The drawback of this study is that the algorithm presented calculates only the similarity between vertices, instead of sub-graphs. As a result, high similarity score of two vertices can produce falsepositively detected design patterns. The matching algorithm used in this thesis, on the other hand, depends on isomorphic sub-graph search and two graphs to find the candidate design patterns in the software project are compared.

Similar to the study of this thesis, an isomorphic graph matching method used to detect design patterns is given in [14]. This approach uses only class diagrams of the GoF design patterns for detecting patterns which might cause false-positive outputs. They do not consider sequence diagrams of patterns where the behavior of pattern lies. In this thesis, it is considered both sequence diagrams and class diagrams. For example, “Class A creates an object of Class B” is a behavior type relation that is taken into consideration. Shortly, the relation set used in this thesis is more specialized in terms of structure and behavior of patterns. Detecting design patterns by using graph matching and Constrain Satisfaction Problem (CSP) search algorithms in an Abstract Semantics Graph (ASG) of a given software project is another method applied in [15]. While they take the entire AST of a given project into the ASG of the project, a high-level model graph by taking only four kinds of nodes and twelve types of relations, which are considered sufficient for detecting the GoF design patterns, is built. This approach is more proper to find design patterns in a more simplified way.

There are also studies in the literature for detecting design patterns in a software project in terms of reverse-engineering methods [5, 6].

PINOT is a tool presented in [5] which allows searching for design patterns based on their structures and then performing “static program analysis”, e.g. “data flow analysis” and “control flow analysis” to detect methods collaboration. As compared to this thesis, there are three basic differences. First, PINOT uses specific keywords to detect design patterns while the study of this thesis remains more generic. For instance, PINOT detects “template method” design pattern by specifically looking up final methods. So, it may fail if a template method design pattern does not contain any final method. Consequently, this thesis decreases the rate of false-positive detected patterns. Second, while PINOT depends on the java compiler (Jikes) for searching patterns, the isomorphic sub-graph search algorithm used in this thesis is independent of any programming language. Third, it is not easy to add new patterns or modify existing ones in PINOT while it is easy to perform such tasks without requiring any coding or compilation in this thesis.

An approach based on static and dynamic analysis of software project’s ASG (Abstract Semantics Graph) is presented in [6]. The detection process of this approach is executed during the run-time of the software by means of log analysis. Therefore, it can only detect patterns that occur at run-time as difference to this thesis where the entire source code at design time is analyzed.

Properties of design patterns are correlated with some of the software metrics in other works. Creating design pattern fingerprints by specifying the roles and metrics of classes is studied in [7]. They reduce the search space by implementing a machine-learning algorithm in their repository.

There also exists another one in which they implemented multi-stage reduction process by using object-oriented software metrics and structural properties to detect design patterns from a software project’s source code [8]. Because they hard-coded rules for detection process, they experimented with only five GoF patterns in their implementation. However, in this thesis, it is worked on higher levels to extract design patterns and this is more flexible. Thus, all of the GoF patterns can be easily experimented.

To the best of current knowledge, the most relevant study to this thesis in terms of building graph model of a source code, is another graph mining approach for detecting frequently used identical sub-structures in a software project by a frequent sub-graph mining method using open-source Parsemis tool [11, 18]. While detecting especially GoF design patterns in a software project and tag them automatically is focused on in this thesis, they detect frequent sub-graph identical structures and then tag them manually. Some new relations like “Class A has the return type of Class B”, “Class A related with its method of Class B” etc. in order to form templates of design patterns properly are also added.

2.2. Software

In order to detect the design patterns that constitute the basis of software projects, several current open-source tools and technologies are used. In the next sections, they are introduced and explained how they are integrated to this thesis.

2.2.1. Subdue

Subdue is a graph-based knowledge analysis tool that detects structural, relational patterns in graph data consisting of entities and relationships.

Subdue performs the minimum description length (MDL) principle, which differs from its contemporaries, to recognize patterns that minimize the number of bits required for describing the input graph after being compressed by the pattern [31]. Sub-graph isomorphism algorithm called sgiso is an efficient algorithm for detecting sub-graphs. Therefore, Subdue is chosen for this work since it contains sgiso as an auxiliary tool. In [32] there are some test results compared with sgiso and other sub-graph detection algorithms. Its results show that sgiso is faster and more accurate in medium-sized projects than other search algorithms.

Subdue gets and produces files in its specific format. Input files are generated automatically by DesPaD according to this format as it is shown in Figure 2.1. Additionally, output files of Subdue are also parsed for visualization in the tool, DesPaD.

```
v 1 Class
v 2 Interface
v 3 Class
e 1 2 related_with_its_method
e 2 2 has_a_method_with_the_return_type_of
e 3 2 implements
e 3 2 related_with_its_method
```

Figure 2.1. An example input file for Subdue

While good results are taken from Subdue in terms of performance in small and medium-sized graphs, big-sized graphs cause bottleneck during the detection operation. At this point, the search space is pruned with a heuristic function explained in Section 3.3, Algorithm 1. It explicitly speeds up the detection process.

2.2.2. Eclipse

DesPaD is developed in Java environment. Eclipse is an open-source software framework written in Java programming language. In its default design, it is a Java Integrated Development Environment (IDE), containing of the Java Development Toolkit (JDT) and compiler (ECJ). Recently, it has become the most prominent and a standard development platform due to its flexibility for most of the Java developers.

In this thesis, some additional libraries, which are ANTLR 4.0 and JFormDesigner-Java Swing GUI Designer for Eclipse, are used. ANTLR, which stands for “Another Tool for Language Recognition”, is defined as “A parser generator that uses LL (a top-down parser for a subset of context-free languages) parsing” in [20]. It is able to generate lexers, parsers, and tree parsers and provide the capability of traversing trees.

The course of grouping characters into words or tokens is called *lexical analysis* or *tokenizing*. The program that tokenizes the input is called a *lexer*. The second step is parsing process by the help of actual parser (language recognizer) and the structure of sentence is noted as tokens. Finally, *a parse tree* or *syntax tree* is built by ANTLR-generated parsers. Figure 2.2 shows the basic data flow of this process.

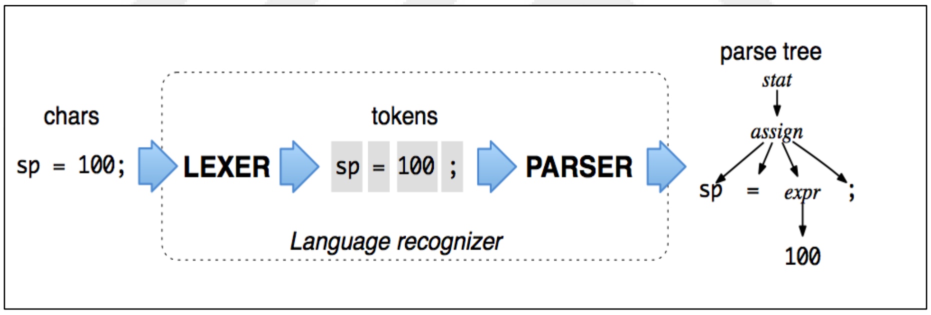


Figure 2.2. Basic data flow of a language recognizer

Building Abstract Syntax Tree (AST) of each class of the given software project, is done by the help of ANTLR library. Afterwards, high-level model graph is formed by compounding and filtering ASTs according to the vertex and edge types described in Section 3, Table 3.1 and Table 3.2.

JFormDesigner is a second plug-in used in Eclipse platform for developing DesPaD tool. It is a convenient GUI designer for Java Swing user interfaces. Its excellent support for some layout components and other controls makes it easy to create well-designed looking forms. It decreases the time spent on hand coding forms. It is powerful and productive for developers [36]. DesPaD’s user-friendly interface comes from the flexibility and power of JFormDesigner.

2.2.3. Graphviz

Graphviz is an open-source graph visualization tool. Graph visualization is a process of showing structural data in the form of abstract graph diagrams and networks.

Graphviz contains some batch layout programs like dot, neato, fdp and twopi [35]. In this thesis, dot layout program is used to plot model graph and detected possible graphs of design patterns to represent them to the architect or developers of the given object-oriented software project. An example of UML-like class-relationship diagram of a tiny source code is seen in Figure 2.3. In this figure, the node labels of C and I stand for Class and Interface respectively. The edge labels are also represented and described in Section 3, Table 3.2.

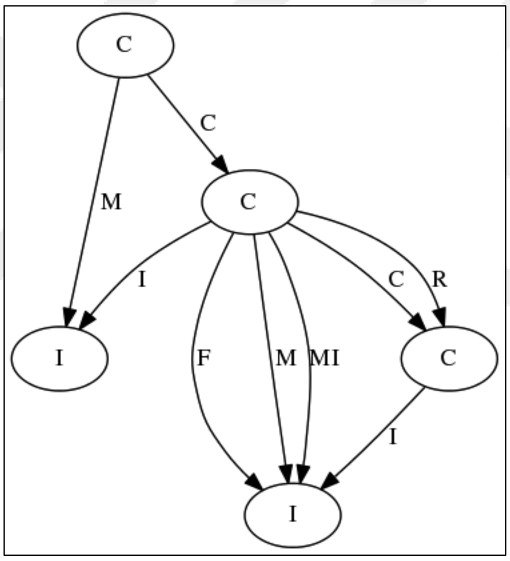


Figure 2.3. An example graph plotted with Graphviz

3. MODEL FOR DESIGN PATTERN DETECTION AND ANALYSIS

The approach of this thesis to detect design patterns consists of three basic steps. First, the source code is analyzed and abstract syntax trees (AST) are extracted out of it. Then, a model graph is built based on these ASTs. Second, templates are generated for all the GoF design patterns. These patterns will be used basically as query items. They are generated only once unless new design patterns are introduced in the literature. Third, the pattern templates are searched in the model graph by using Subdue’s sub-graph mining algorithm called sgiso. The overview of design pattern detection architecture is seen on Figure 3.1.

A fully automated, java based design pattern detection tool called DesPaD (Design Pattern Detector) is developed to execute all the steps given above. It is fast, convenient to use and targets at finding design patterns with high precision. DesPaD is freely available at Github [29].

3.1. Model Graph Creation

In this step, a high-level graph representation of an object-oriented software project’s source code is generated. A software project is represented as a simple labeled and directed graph (G).

Formally, a graph is defined as “A formation by vertices and edges connecting the vertices” [19]. To establish a common understanding, definitions of graph related notions used throughout this thesis are given as follows:

Software Model Graph (G): G = (V, E, L_e, L_v) is a labeled directed graph. V is a cluster of vertices or nodes, E is a cluster of edges or arcs, L_v is a cluster of labels for the vertices and L_e is a cluster of labels for the edges [11].

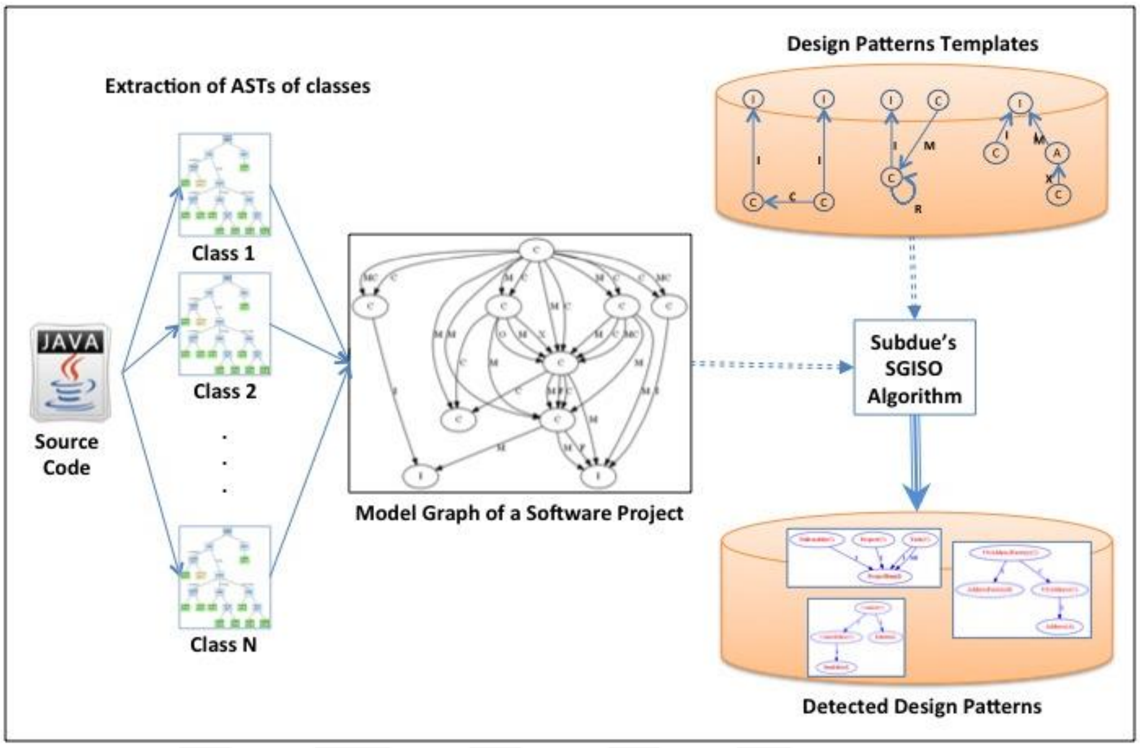

Extraction of ASTs of classes
Class 1
Class 2
Class N
Source Code
Model Graph of a Software Project
Design Patterns Templates
Subdue's SGISO Algorithm
Detected Design Patterns

Figure 3.1. Overview of DesPaD's pattern detection architecture

Sub-graph: It is a subset graph of another graph. Its detailed definition is given in Section 1.2.

Isomorphic sub-graph: Two identical sub-graphs are said to be isomorphic sub-graphs. Its detailed definition is given in Section 1.2.

The vertices in the model graph G, generated out of the source, are classes, template classes, abstract classes or interfaces. The edges of G include the specific relations of inheritance, aggregation, association or composition properties used commonly in object-oriented programming.

Vertices are labeled in the model graph generated out of a source code by using the abbreviations as shown in Table 3.1.

Table 3.1. Vertex labels and types

Vertex Label	Entity Type
C	Class
I	Interface
T	Template Class
A	Abstract Class

As the final goal of this work is to detect the relations of GoF design patterns in a source code, the class diagrams and collaborations (also called sequence diagrams) are analyzed within every GoF design pattern [4]. As a result of this analysis, relations are identified as listed in Table 3.2.

There are two types of relations in the list of Table 3.2. One of them contains the high level behavioral relations extracted from sequence diagrams like “Class A calls method of Class B”, “Class A creates an object of Class B” and “Class A has the return type of Class B”. The other type consists of the relations which are extracted from class diagrams like “Class A extends Class B” or “Class implements Class B” etc.

The building process of the model graph starts with generating the abstract syntax tree of each class in the given software. ANTLR [20], which is an open-source Java library, is used for generating ASTs. ANTLR library is described in detail in Section 2.2.2.

Entire java language grammar is available as BNF (Backus Normal Form) diagrams [21]. DesPaD uses these BNF diagrams to detect relations listed in Table 3.2. For instance, the inheritance relations like “extends” and “implements” in class declaration are detected by using the BNF diagram in Figure 3.2.

Table 3.2. Edge labels, relations and types

Edge Label	Relation Description*
X	Class A extends Class B
I	Class A implements Class B
C	Class A creates object of Class B
O	Class A overrides a method of Class B
MC	Class A calls a method of Class B
F	Class A has the field type of Class B
MR	Class A has a method with the return type of Class B
ML	Class A has a method that defines a local variable with the type of Class B
MI	Class A has a method that has an input parameter with the type of Class B
M	Class A has related with its method of Class B
R	Class A has the return type of Class B
G	Class A uses Class B in a generic type declaration

** Class A is directed to Class B.*

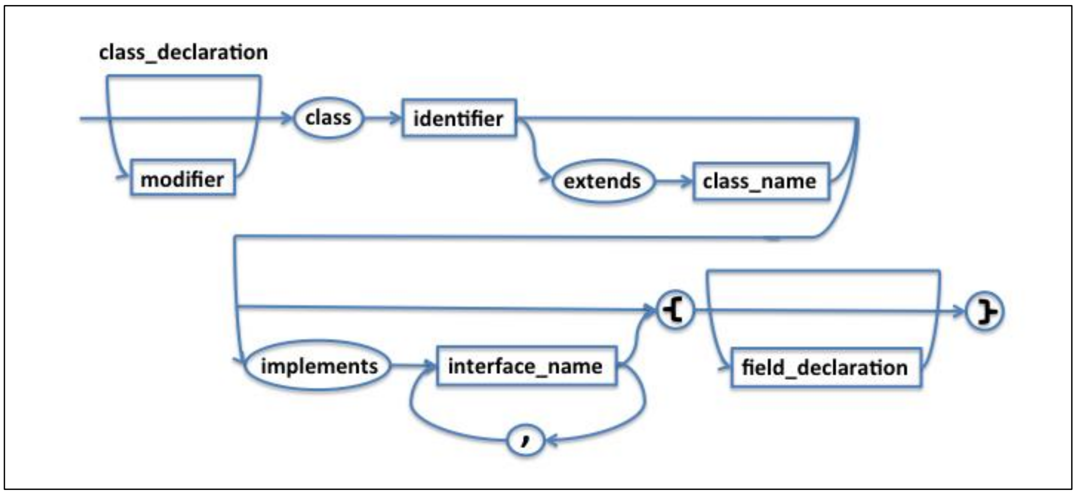


Figure 3.2. BNF Diagram of class declaration in Java Grammar Language

In some design patterns like iterator or singleton, the internal libraries of java are to be used for building that pattern; otherwise it is difficult to detect those patterns in the source code. Thus, it is meaningful to add internal classes and their relations to the model graph created in related patterns. This option is presented on the GUI of the DesPaD tool as ‘Include inner classes’ checkbox.

As a result, a model graph for sub-graph mining process is created similar to the one in Figure 3.3. DesPaD prepares and creates the model graph in a file formatted for the open-source sub-graph mining tool, Subdue.

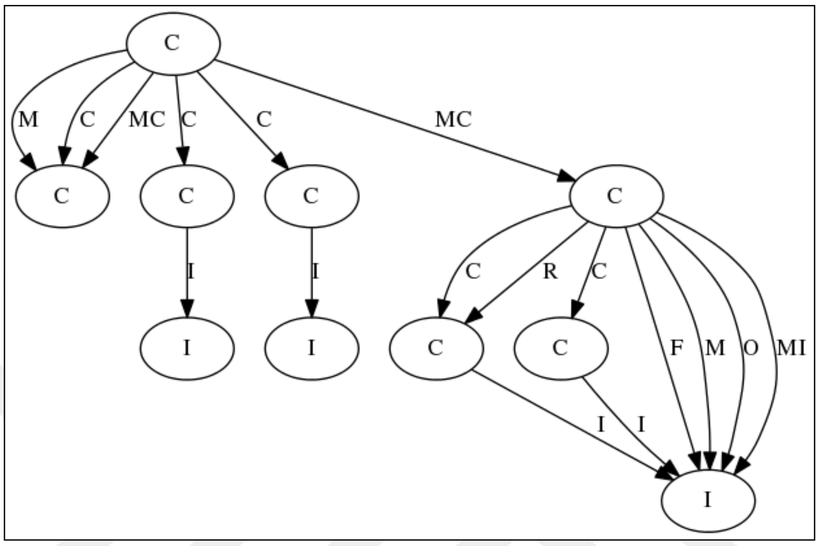


Figure 3.3. An example model graph

3.2. Design Pattern Template Creation

After building the model graph as the search space, the goal is to search for sub-graphs that might represent the GoF design patterns. To achieve this, the class and sequence diagrams of all 23 GoF design patterns are analyzed and template graphs are generated to represent subgraphs for each of them. An example template that was generated for the bridge design pattern is seen in Figure 3.4.

As seen in Figure 3.4, vertices were tagged with *1*, *M* and *N*. *1* means that the vertex and its edges occur only once. *M* and *N* mean that the vertex and its edges can occur more than once. DesPaD determines the maximum values for M and N by counting the numbers of times a node has a specific relation. That is, for the bridge pattern given in Figure 3.4, maximum numbers of times any class in the entire source code was extended or implemented are assigned to M and N, respectively. Afterwards, all possible design patterns templates are generated to cover M x N possible combinations that might represent a bridge pattern. For example, according to the bridge pattern template graph in Figure 3.4, if *M* is 11 and *N* is 5, this means that a class was extended maximum 11 times and an interface was implemented maximum 5 times in the target source code. Accordingly, the number of the bridge pattern template graphs that will be generated is 55. These 55 input files are saved for sub-graph mining tool, Subdue.

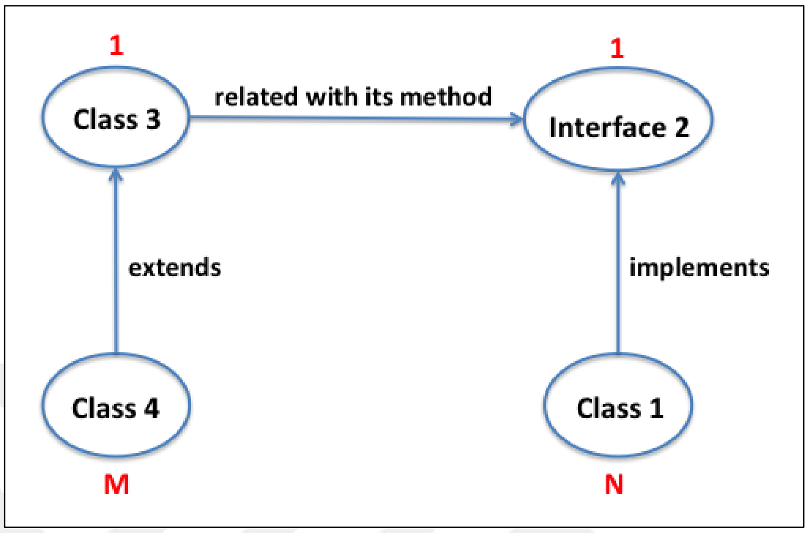


Figure 3.4. Template of Bridge Design Pattern

3.3. Detecting Design Patterns

After having generated the model graph of the software and template graphs for every design patterns, it is time to execute sub-graph mining search. To do this, an isomorphic sub-graph mining algorithm is used. The algorithm is called “sgiso” and provided by the open-source graph-mining tool, Subdue [17].

The algorithm for detecting design patterns is given in Algorithm 1. Maximum numbers for *M* and *N* are given as input to the algorithm. Apparently, it will be time and resource consuming to execute all combinations of the candidate templates against the isomorphic sub-graph search tool. For example, if one considers the bridge pattern in Figure 3.5, if *M* is 11 and *N* is 5, there would be 55 combinations to run the isomorphic search against. Instead, the algorithm stops trying after some value *i*, if the sub-graph search returns nothing for *i+1* (See Algorithm 1, lines 5, 11, 19). That is when M is 11, if a specific class was extended four times, after the fifth iteration, there is no point for going sixth iteration and more.

After the algorithm is executed, there might be overlapping sub-graphs in the output list. Overlapped sub-graphs are eliminated accordingly. Finally, found design patterns can be visualized by DesPaD. To achieve this, DesPaD uses the open-source GraphViz application [22]. An example bridge pattern extracted from Java AWT 1.3 project is visualized as seen in Figure 3.6.

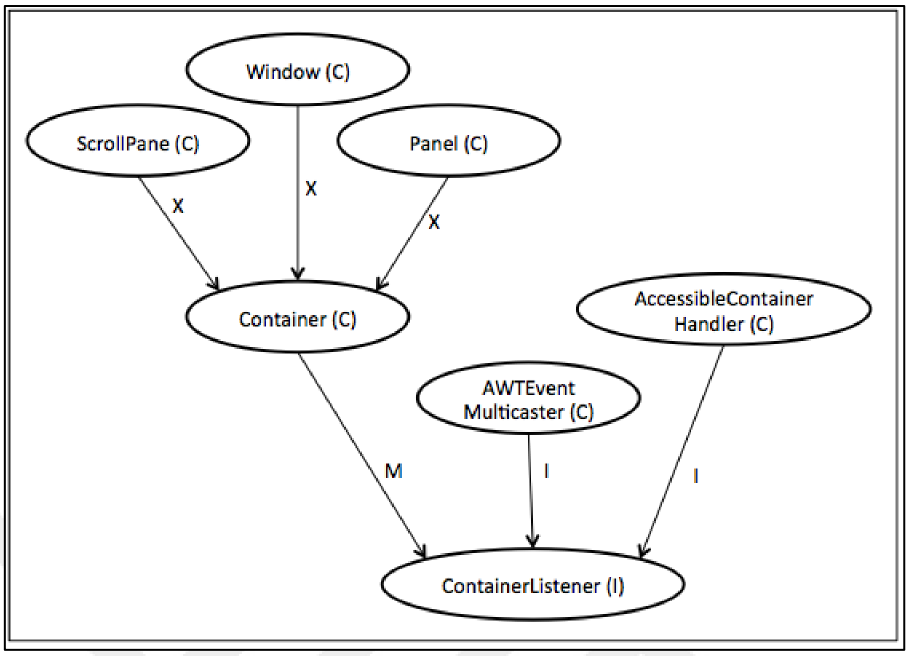


Figure 3.5. An example bridge pattern extracted from Java AWT 1.3

Algorithm 1. Detection of Design Patterns

```
Data: Number of relations of the given template {$M_i$}, {$N_j$} ($M \geq N$);
      Generated candidate input files input_file[x][y];
foreach x ∈ {$M_i$} do
/* After running sub-graph isomorphism algorithm(sgiso) in Subdue, we get output files in outputs[]. */
  execute sgiso on input_file[x][0];
  add output of sgiso to outputs[];
if no output exists then
    break;
  end
  foreach y ∈ {$N_j$} do
    execute sgiso on input_file[x][y];
    add output of sgiso to outputs[];
  if no output exists then
        break;
    end
  end
end
foreach y ∈ {$N_j$} do
  execute sgiso input_file[0][y];
  add output of sgiso to outputs[];
  if no output exists then
    break;
  end
end
```

3.4. GoF Design Patterns

The Applied Java Patterns book [23, 37] comes with some example source codes for each GoF pattern. Generic templates of each GoF pattern are tested in these small-sized source codes.

For each pattern, the templates generated in DesPaD are defined as shown in this section. The model graph of source code’s demo applications of ‘Applied Java Patterns’ is plotted in DesPaD as demonstrated for each pattern. According to the GoF book [4], design patterns fall into three classes based on their purposes. These are creational patterns, structural patterns and behavioral patterns.

3.4.1. Creational Design Patterns

In the literature it is stated as “Creational patterns are related with the process of object creation” [4]. This class of patterns includes abstract factory, builder, factory method, prototype and singleton.

Abstract Factory: According to the literature this pattern is described as “Abstract Factory supports an interface for creating families of related or dependent objects without determining their concrete classes” [4].

When a developer wants an autonomous system in which its products are formed, composed and showed, Abstract Factory is a good alternative. This pattern does also supports a system that it is arranged with one of multiple families of products. The capability of demonstrating only the interfaces of the classes is also possible in this pattern.

Figure 3.6 shows the template of Abstract Factory design pattern, which is defined in DesPaD.

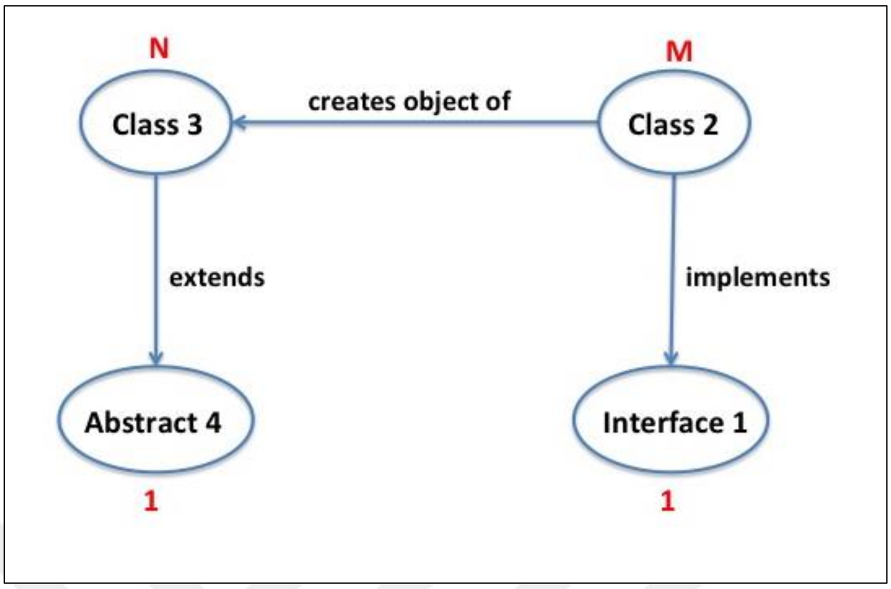


Figure 3.6. Template of Abstract Factory Design Pattern

Figure 3.7 shows the created model graph in the example source code in Applied Java Patterns book. The template of Abstract Factory Design Pattern is sought in this model graph.

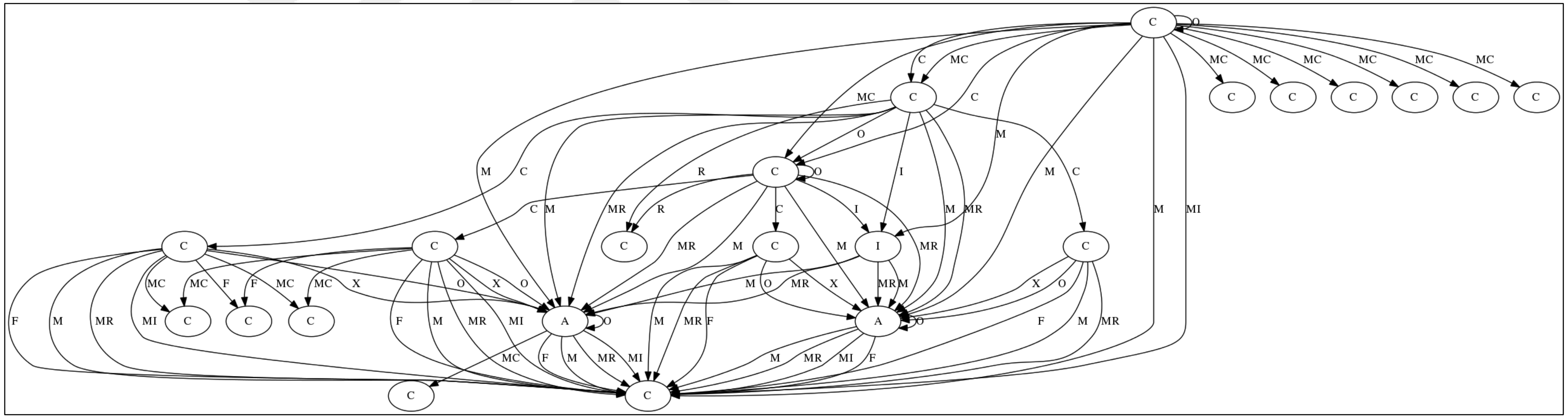


Figure 3.7. Graph Model of AJP - Abstract Factory example source code

As a result of the search, the detected pattern of Abstract Factory is demonstrated in Figure 3.8.

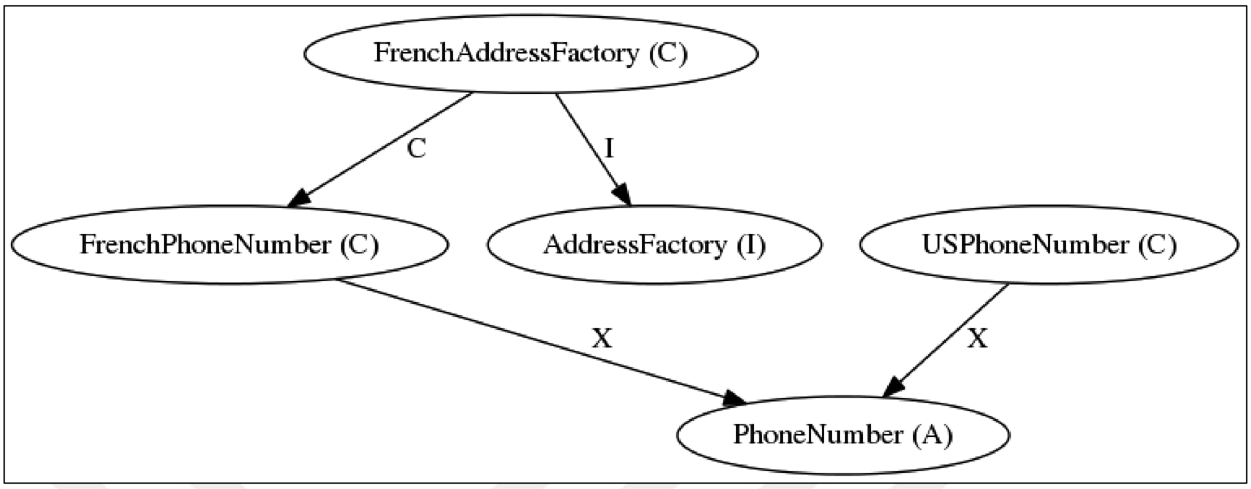


Figure 3.8. Abstract Factory Pattern detected in AJP

Builder: According to the literature this pattern is described as “Builder detaches the construction of a complex object from its representation so that the same construction procedure can create various representations” [4].

When the method in a formed complex object must be autonomous of the parts forming the object, Builder pattern is beneficial. In this pattern, different samples for the object may be shown by the help of the construction process.

Figure 3.9 shows the template of Builder design pattern, which is defined in DesPaD.

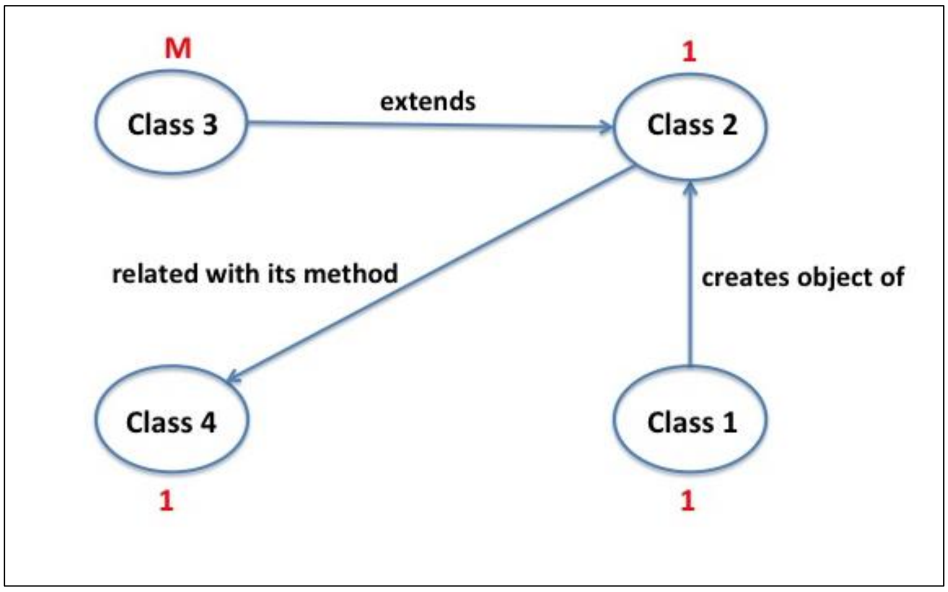


Figure 3.9. Template of Builder Design Pattern

Figure 3.10 shows the created model graph in the example source code in Applied Java Patterns book. The template of Builder Design Pattern is sought in this model graph.

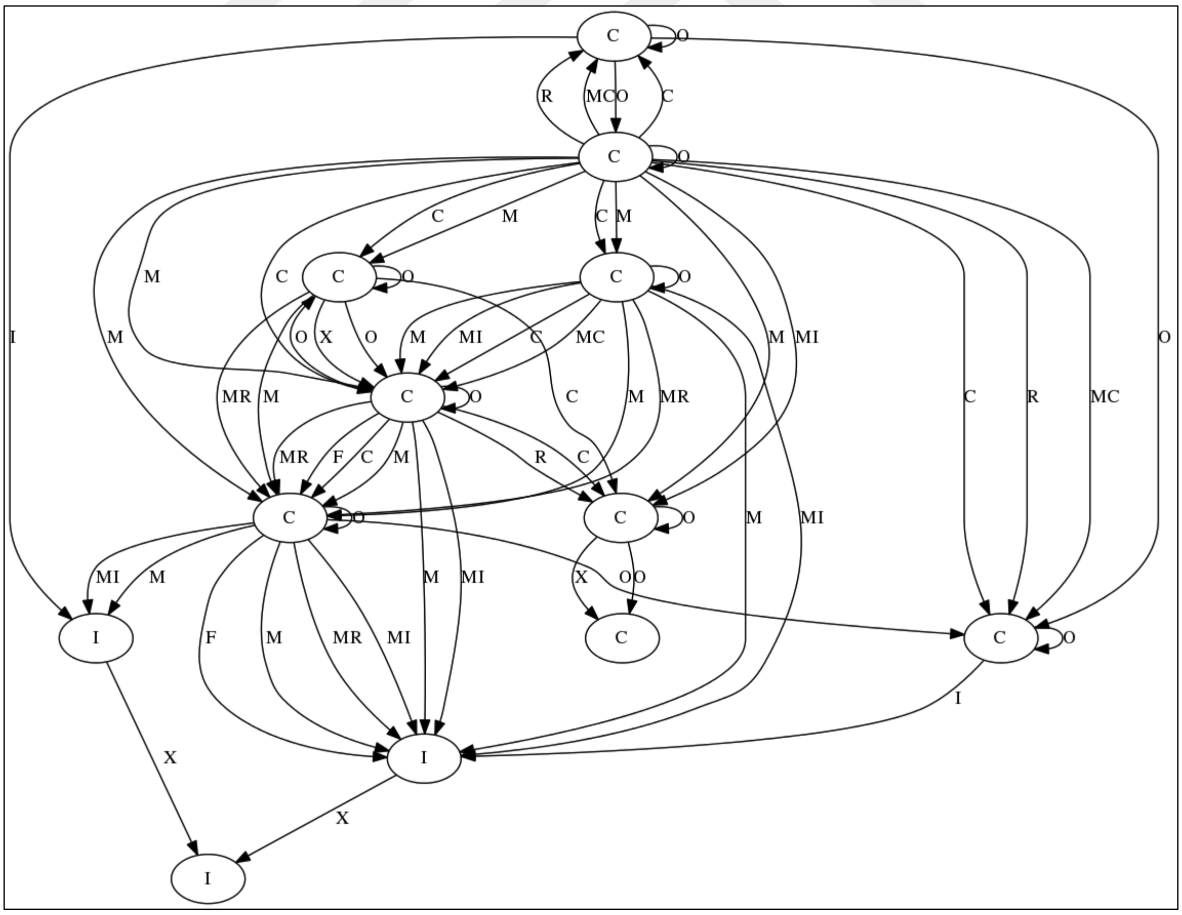


Figure 3.10. Graph Model of AJP - Builder example source code

As a result of the search, the detected pattern of Builder is demonstrated in Figure 3.11.

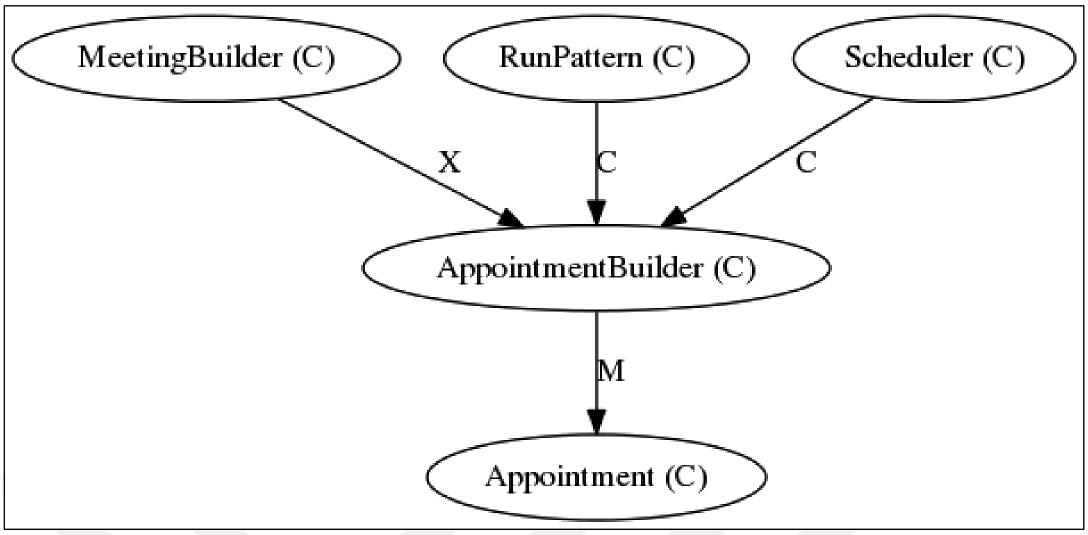


Figure 3.11. Builder Pattern detected in AJP

Factory Method: According to the literature this pattern is described as "Factory Method defines an interface for creating an object, but lets subclasses resolve which class to initiate" [4].

When a developer wants to create a class that cannot assume the class of objects it should create, the Factory Method design pattern may be chosen as the solution. The Factory Method is useful when a class wants its subclasses to determine the objects it creates. Moreover, when one wants to build a system in which responsibility is delegated by classes to one of some helper subclasses, this design pattern should be used as an alternative.

Figure 3.12 shows the template of Factory Method design pattern, which is defined in DesPaD.

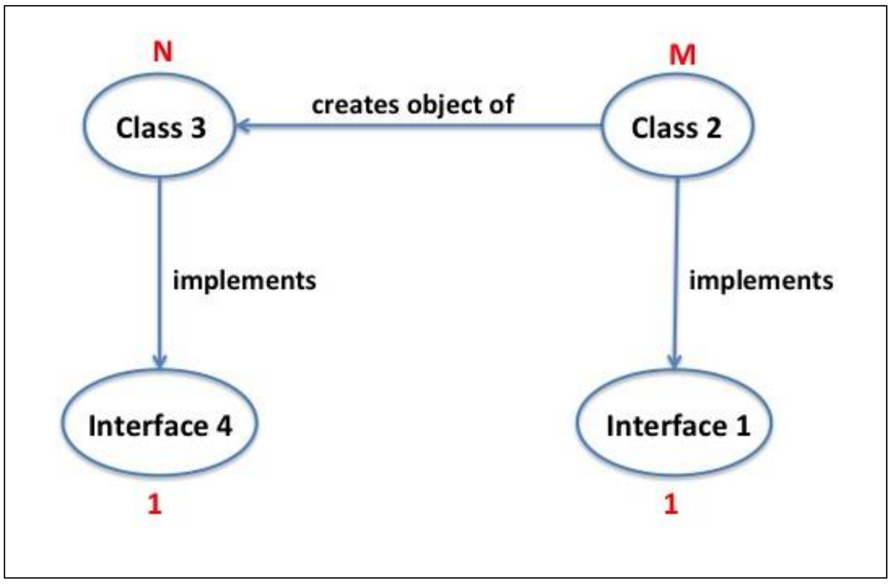


Figure 3.12. Template of Factory Method Design Pattern

Figure 3.13 shows the created model graph in the example source code in Applied Java Patterns book. The template of Factory Method Design Pattern is sought in this model graph.

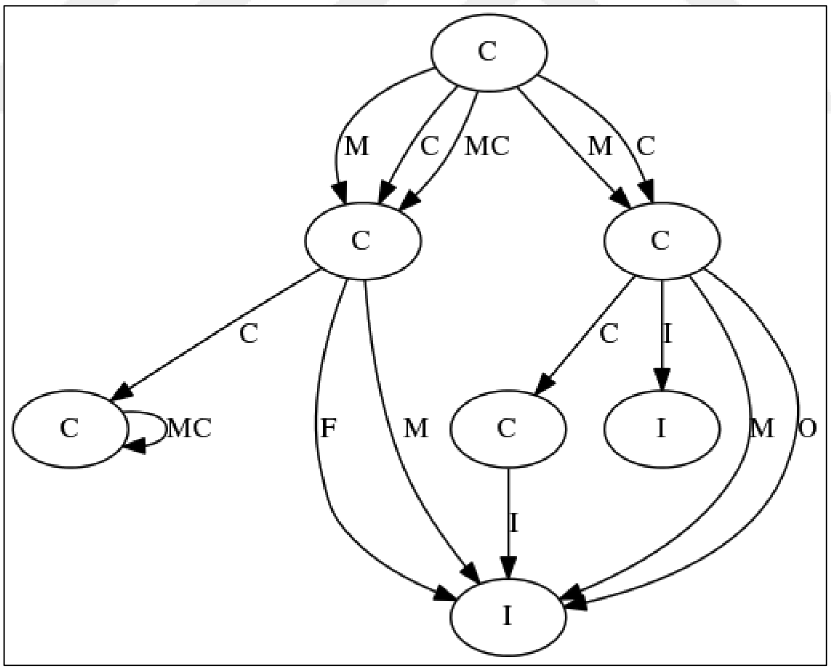


Figure 3.13. Graph Model of AJP - Factory Method example source code

As a result of the search, the detected pattern of Factory Method is demonstrated in Figure 3.14.

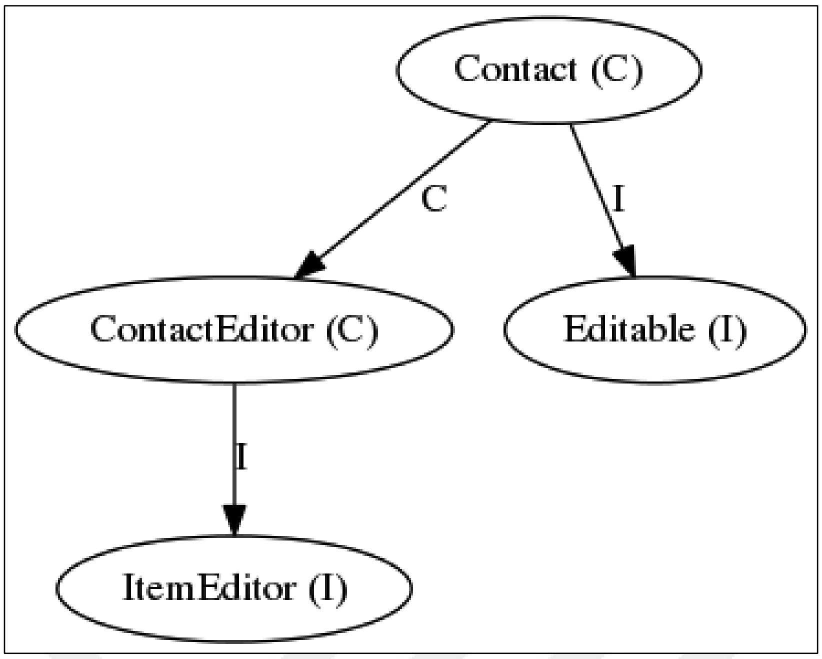


Figure 3.14. Factory Method Pattern detected in AJP

Prototype: According to the literature this pattern is described as “Prototype determines the kinds of objects to create using a prototypical example, and create new objects by copying this prototype” [4].

If a system must not be dependent on the way its products are formed, composed and showed, prototype pattern may be used. Also using this pattern is a good choice when the developer wants to determine the exemplified classes at run-time, such as, by dynamic loading.

Figure 3.15 shows the template of Prototype design pattern, which is defined in DesPaD.

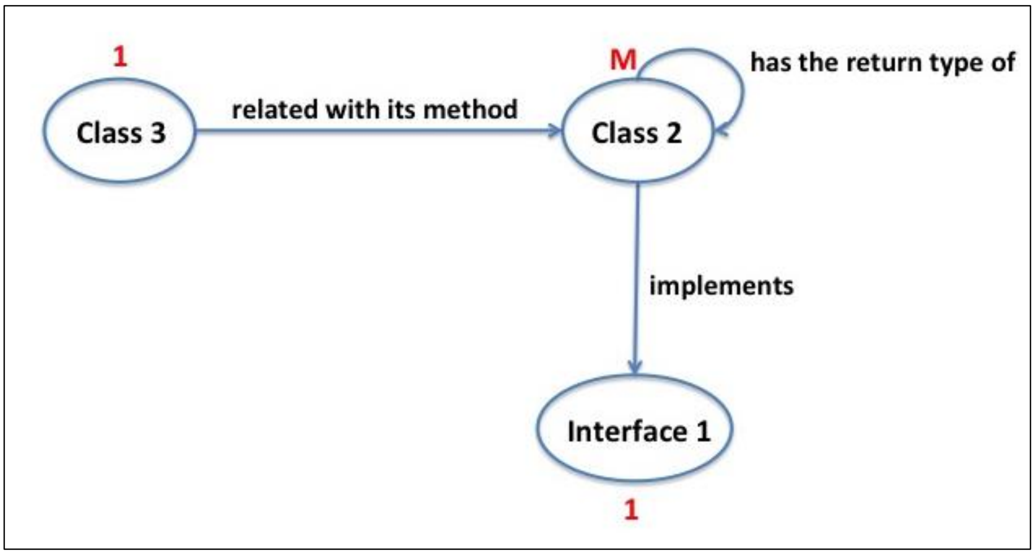


Figure 3.15. Template of Prototype Design Pattern

Figure 3.16 shows the created model graph in the example source code in Applied Java Patterns book. The template of Prototype Design Pattern is sought in this model graph.

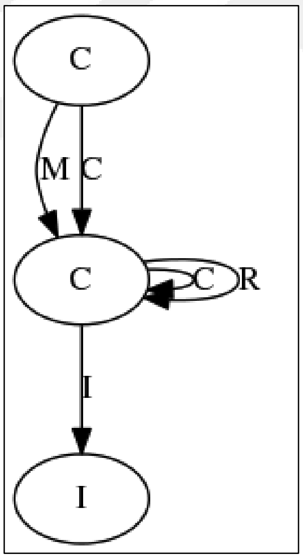


Figure 3.16. Graph Model of AJP - Prototype example source code

As a result of the search, the detected pattern of Prototype is demonstrated in Figure 3.17.

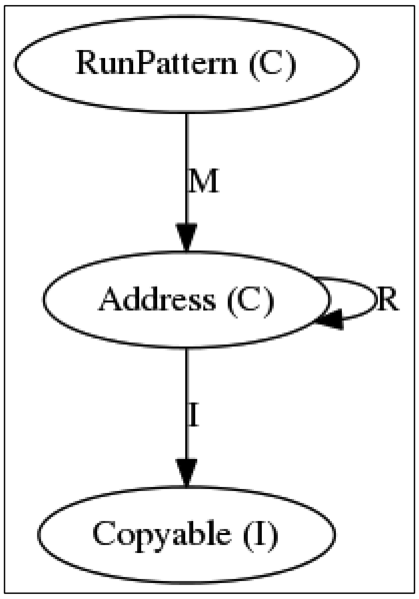


Figure 3.17. Prototype Pattern detected in AJP

Singleton: According to the literature this pattern is described as “Singleton assures that a class has only one instance, and supports a global point of access to it” [4].

The developer may choose to apply the Singleton pattern when only one instance of a class should occur and this instance is reachable to clients only from a point of access. Also this pattern is useful when the unique instance of a class wants to be extensible by sub-classing and an extended instance should be implemented by clients without any change in their source code.

Figure 3.18 shows the template of Singleton design pattern, which is defined in DesPaD.

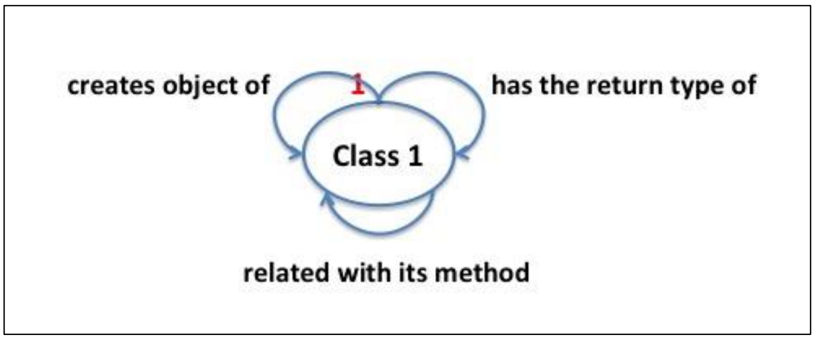


Figure 3.18. Template of Singleton Design Pattern

Figure 3.19 shows the created model graph in the example source code in Applied Java Patterns book. The template of Singleton Design Pattern is sought in this model graph.

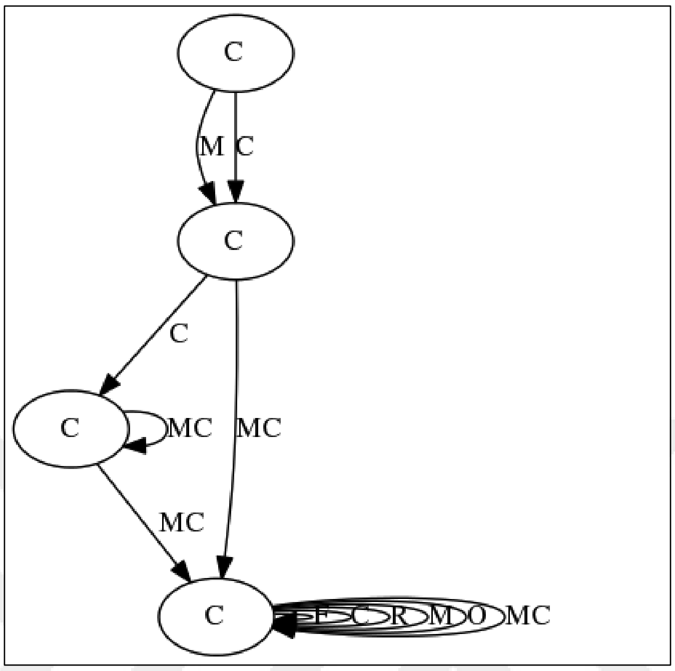


Figure 3.19. Graph Model of AJP - Singleton example source code

As a result of the search, the detected pattern of Singleton is demonstrated in Figure 3.20.

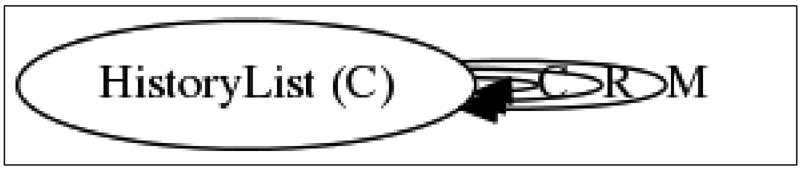


Figure 3.20. Singleton Pattern detected in AJP

3.4.2. Structural Design Patterns

In the literature it is stated as “Structural patterns handle the composition of classes or objects” [4]. This class of patterns includes adapter, bridge, composite, decorator, facade, flyweight and proxy.

Adapter: According to the literature this pattern is described as “Adapter alters the interface of a class into another interface clients expect and it also allows classes to work together in spite of incompatible interfaces” [4].

While an interface of a class is not enough for existing requirements, the solution comes up with Adapter design pattern. If the problem of creating a reusable class that collaborates with classes which do not have compatible interfaces occurs, Adapter is the helper pattern for the developers.

Figure 3.21 shows the template of Adapter design pattern, which is defined in DesPaD.

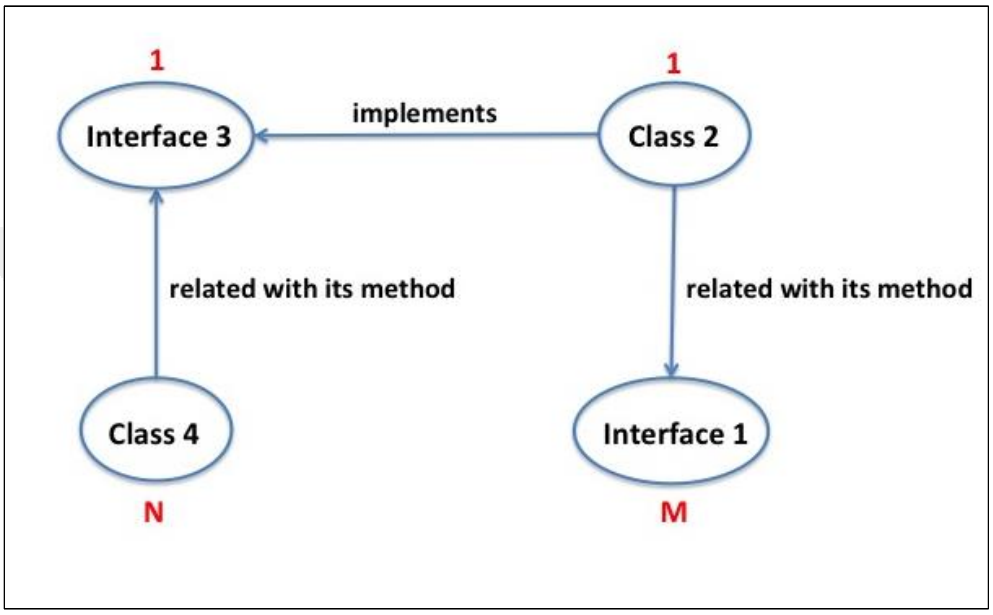


Figure 3.21. Template of Adapter Design Pattern

Figure 3.22 shows the created model graph in the example source code in Applied Java Patterns book. The template of Adapter Design Pattern is sought in this model graph.

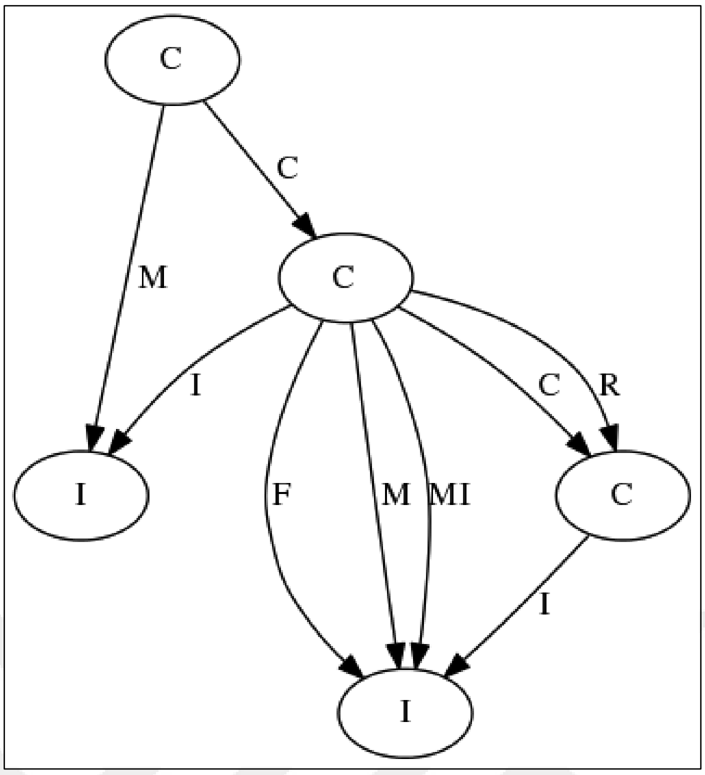


Figure 3.22. Graph Model of AJP - Adapter example source code

As a result of the search, the detected pattern of Adapter is demonstrated in Figure 3.23.

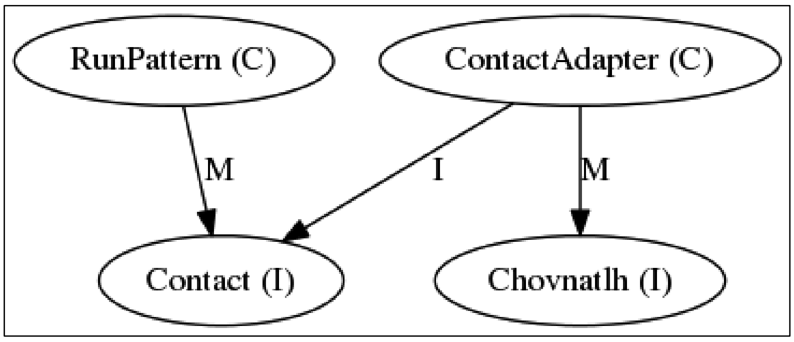


Figure 3.23. Adapter Pattern detected in AJP

Bridge: According to the literature this pattern is described as “Bridge decouples an abstraction from its execution so that the two can vary independently” [4].

In design stage of a software project, if a developer has a problem of preventing a permanent bond between an abstraction and its implementation, Bridge design pattern may be considered as a good solution. For instance, the pattern may be used when the implementation should be switched at run-time of the program. In Bridge pattern, both the abstractions and their implementations ought to extend from subclassing, that is, it lets you link the different abstractions and implementations. If a developer wants to hide the implementation of an abstraction ultimately from clients, Bridge pattern is the one he or she needs.

Bridge pattern also helps developers to build a class hierarchy displaying the need for splitting an object into two sections. Rumbaugh uses the term “nested generalizations” [40] to refer to such class hierarchies. On the other hand, sometimes one may want to share an implementation among many objects (perhaps using reference counting), and the client should not recognize this situation. In such a case Bridge is used. A simple example is Coplien’s String class [41], where multiple objects can share the same string representation.

Figure 3.24 shows the template of Bridge design pattern, which is defined in DesPaD.

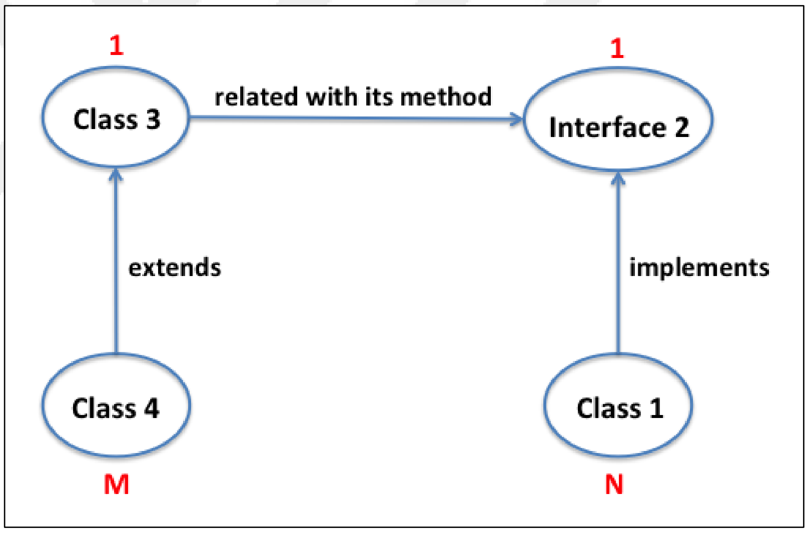


Figure 3.24: Template of Bridge Design Pattern

Figure 3.25 shows the created model graph in the example source code in Applied Java Patterns book. The template of Bridge Design Pattern is sought in this model graph.

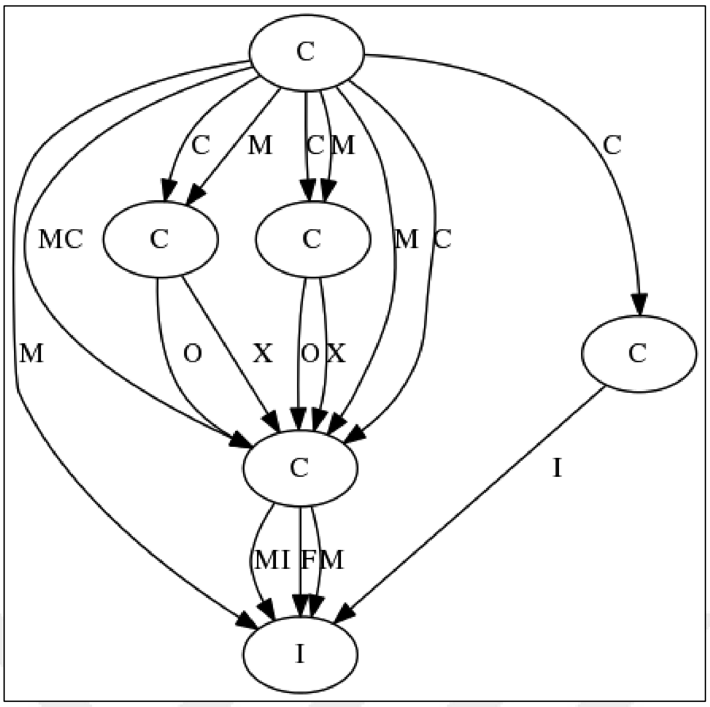


Figure 3.25. Graph Model of AJP - Bridge example source code

As a result of the search, the detected pattern of Bridge is demonstrated in Figure 3.26.

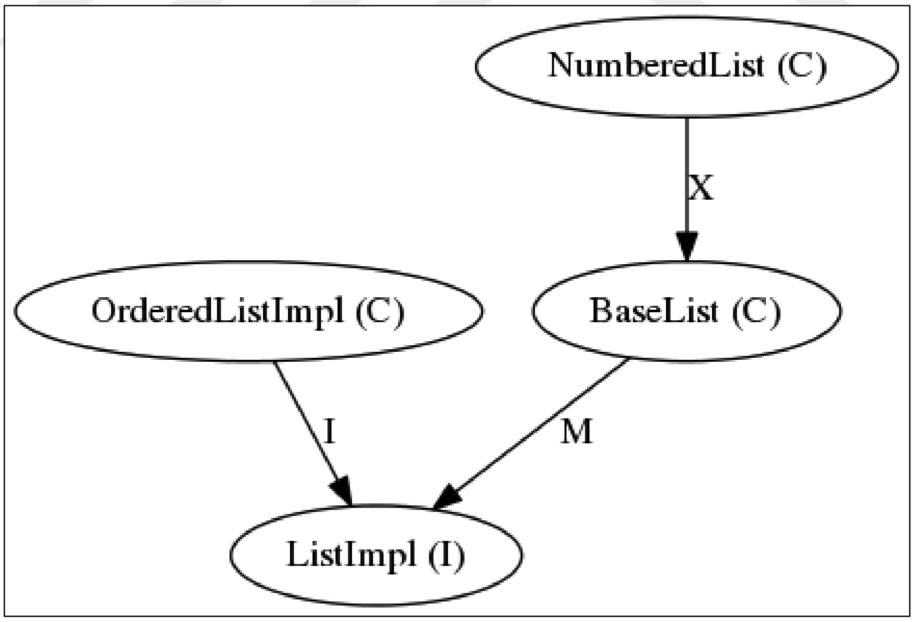


Figure 3.26. Bridge Pattern detected in AJP

Composite: According to the literature this pattern is described as “Composite design pattern composes objects into tree structures to demonstrate part-whole hierarchies and it also allows clients to treat individual objects and compositions of objects uniformly” [4].

While demonstrating part-whole hierarchies of objects in object-oriented programming language, some developers are in trouble. Composite design pattern is for them. It also supports clients not to be aware of the differences between individual objects and object compositions. In this pattern, all objects are treated, without any exception, in the composite structure.

Figure 3.27 shows the template of Composite design pattern, which is defined in DesPaD.

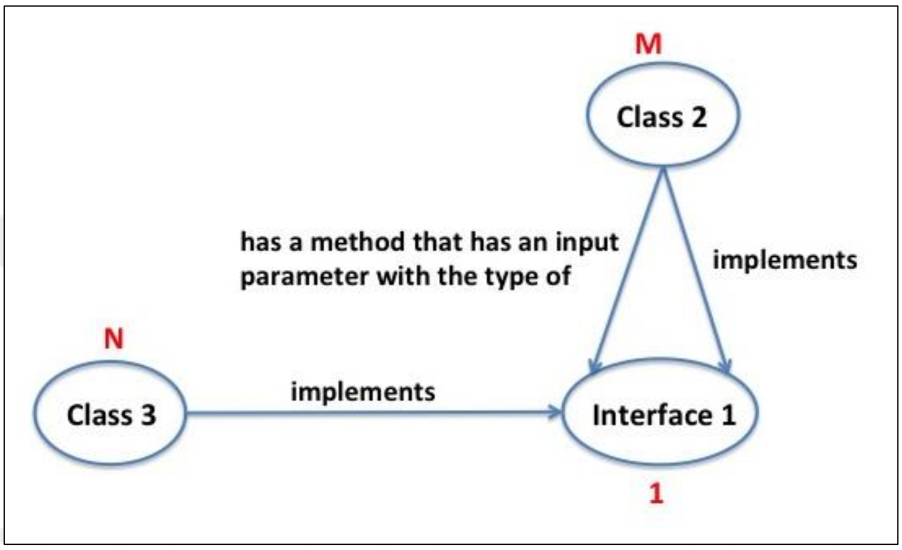


Figure 3.27. Template of Composite Design Pattern

Figure 3.28 shows the created model graph in the example source code in Applied Java Patterns book. The template of Composite Design Pattern is sought in this model graph.

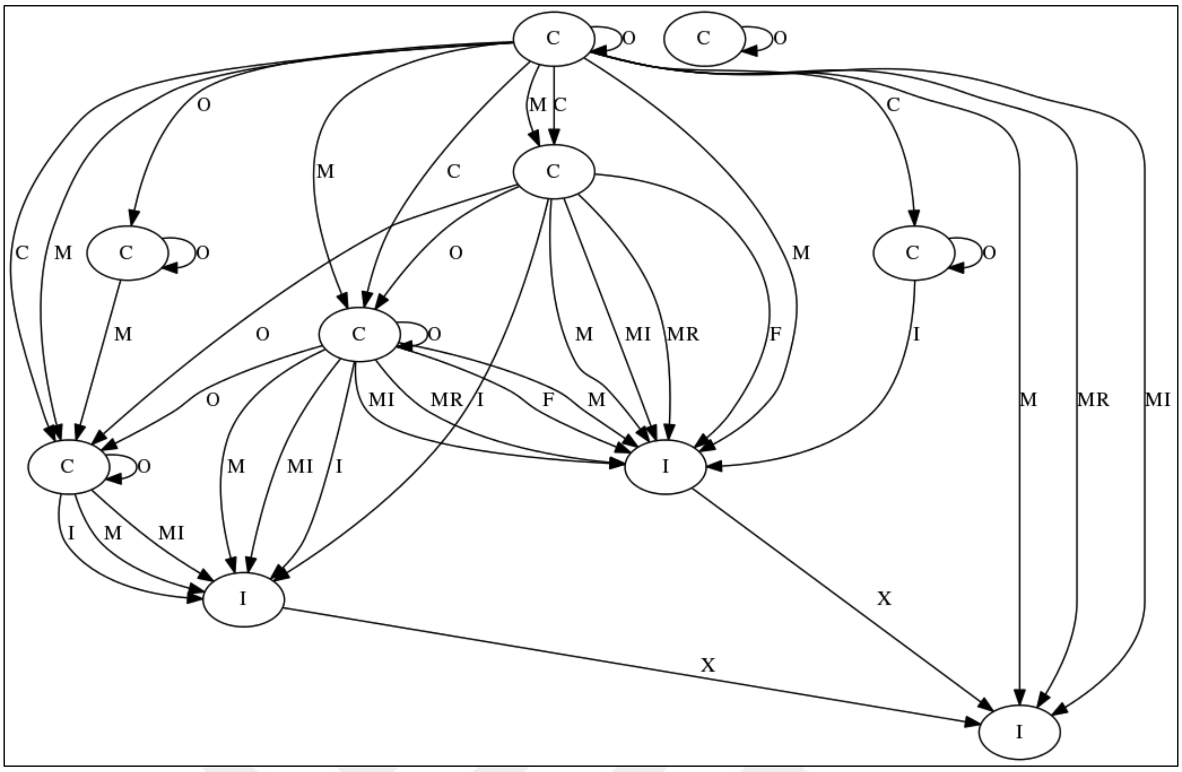


Figure 3.28. Graph Model of AJP - Composite example source code

As a result of the search, the detected pattern of Composite is demonstrated in Figure 3.29.

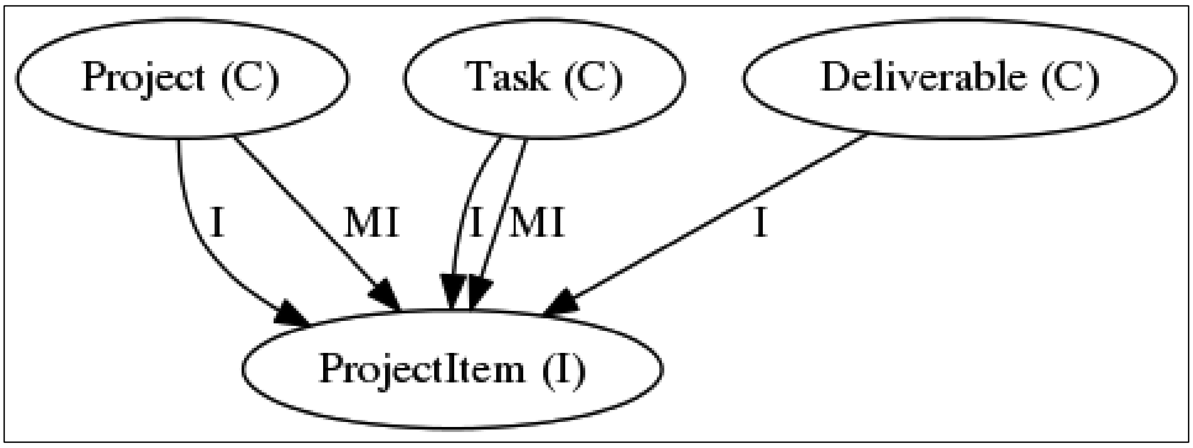


Figure 3.29. Composite Pattern detected in AJP

Decorator: According to the literature this pattern is described as “Decorator affixes additional responsibilities to an object in a dynamic way and provides a flexible alternative to subclassing for enhancing functionality” [4].

In Decorator pattern, it is easy to dynamically add responsibilities to individual objects without affecting other objects. Moreover, withdrawing some responsibilities from a class hierarchy is also possible. The other capability of this pattern is extending subclassing when it is impractical.

Figure 3.30 and Figure 3.31 show the templates of Decorator design pattern, which are defined in DesPaD.

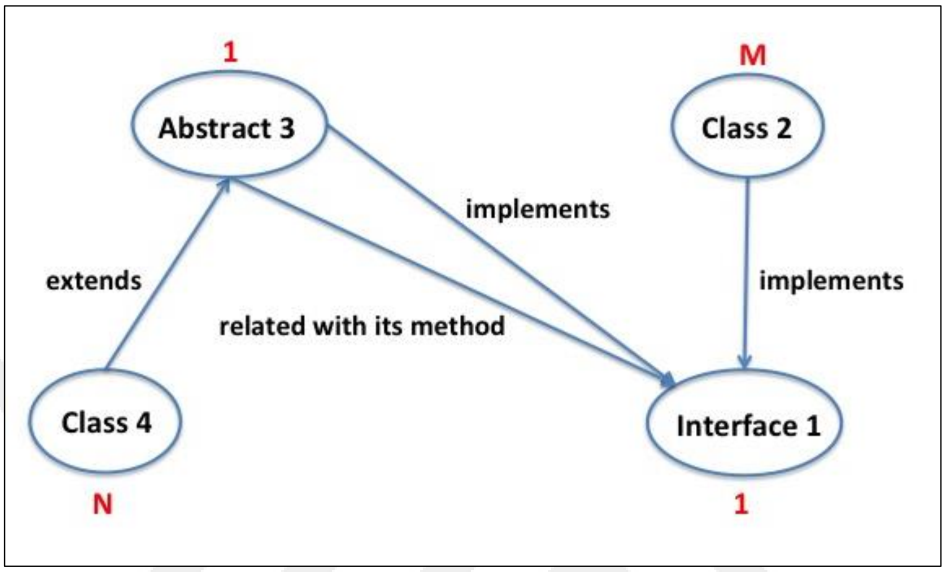


Figure 3.30. Template-1 of Decorator Design Pattern

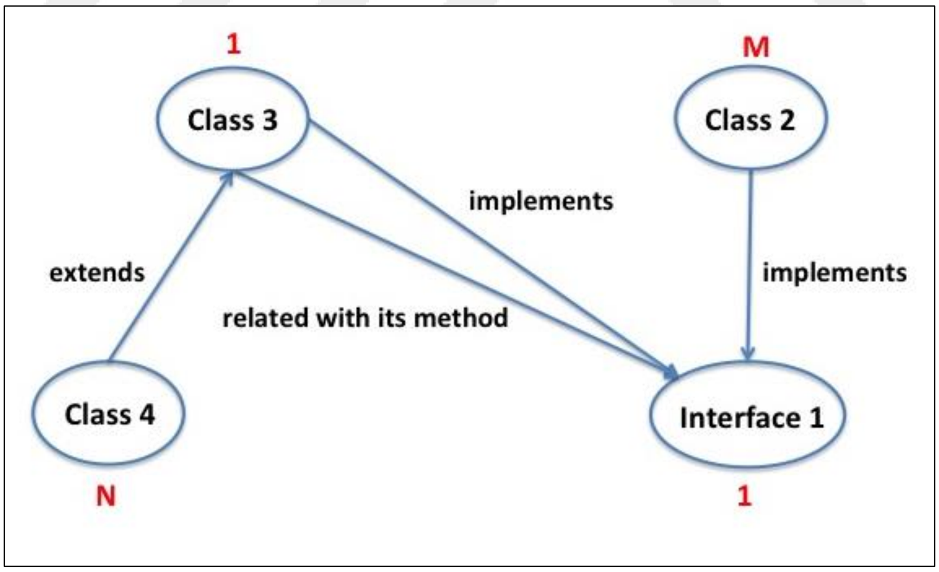


Figure 3.31. Template-2 of Decorator Design Pattern

Figure 3.32 shows the created model graph in the example source code in Applied Java Patterns book. The templates of Decorator Design Pattern are sought in this model graph.

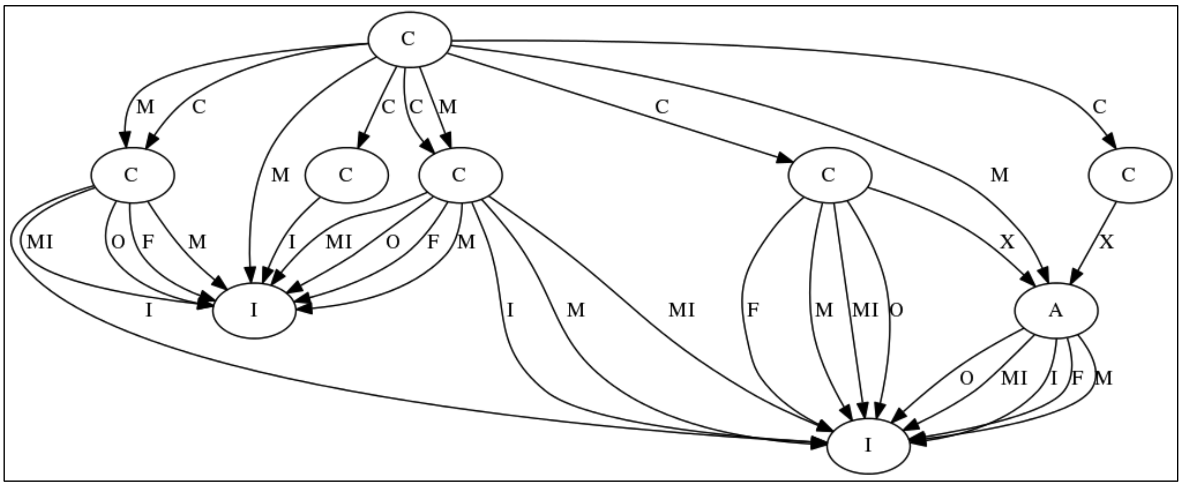


Figure 3.32. Graph Model of AJP - Decorator example source code

As a result of the search, the detected pattern of Decorator is demonstrated in Figure 3.33.

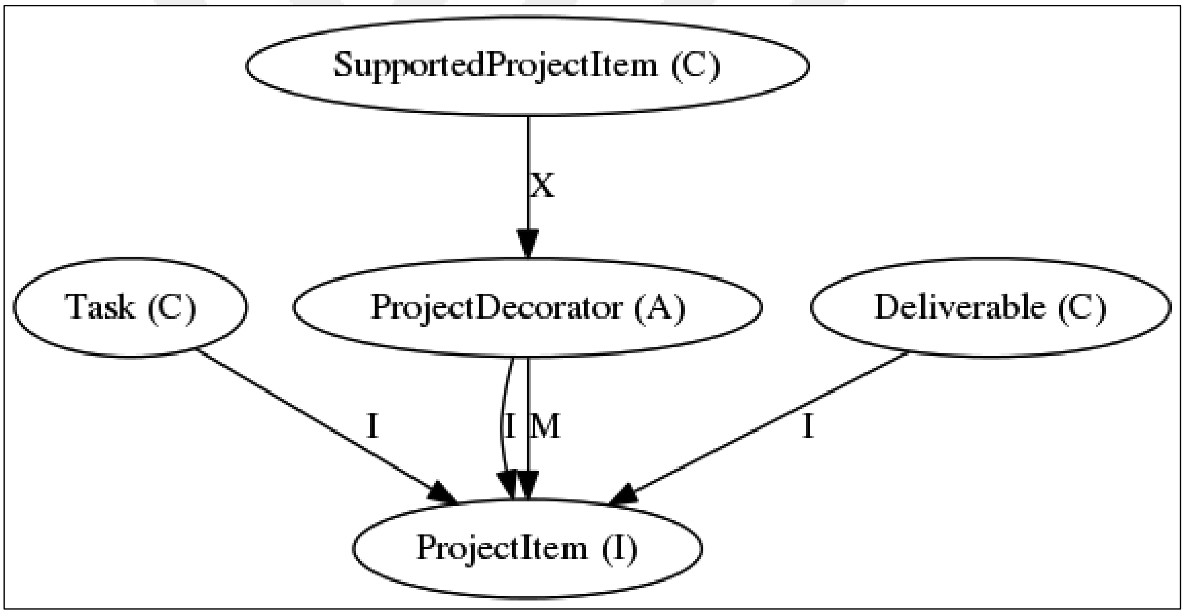


Figure 3.33. Decorator Pattern detected in AJP

Facade: According to the literature this pattern is described as “Facade arranges a unified interface to a set of interfaces in a subsystem and describes a higher-level interface that makes the subsystem more convenient to use” [4].

In object-oriented design, when one wants to supply a moderate interface to a complex system, Facade pattern may be used. By the help of this pattern, the subsystem becomes easier to re-use and customize; on the other hand, it becomes harder to implement for clients that don’t require customizing it. A facade can provide a simple default view of the subsystem, which may be considered as proper for most of the clients.

Clients and the usage of classes of an abstraction may be dependent on each other in many ways. In this case, decoupling the subsystem from clients and other subsystems is proposed by the Facade pattern. Additionally, a layered-architecture of subsystems is built by this pattern. A Facade pattern may be an entry point to each subsystem level.

Figure 3.34 shows the template of Facade design pattern, which is defined in DesPaD.

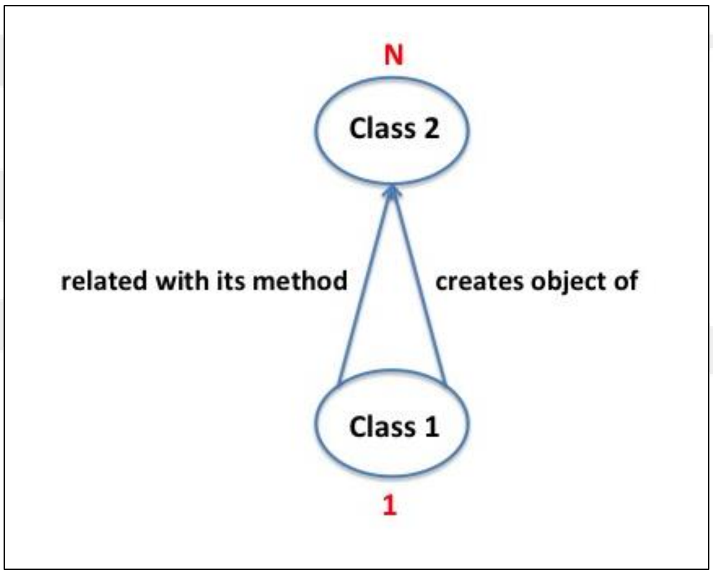


Figure 3.34. Template of Facade Design Pattern

Figure 3.35 shows the created model graph in the example source code in Applied Java Patterns book. The template of Facade Design Pattern is sought in this model graph.

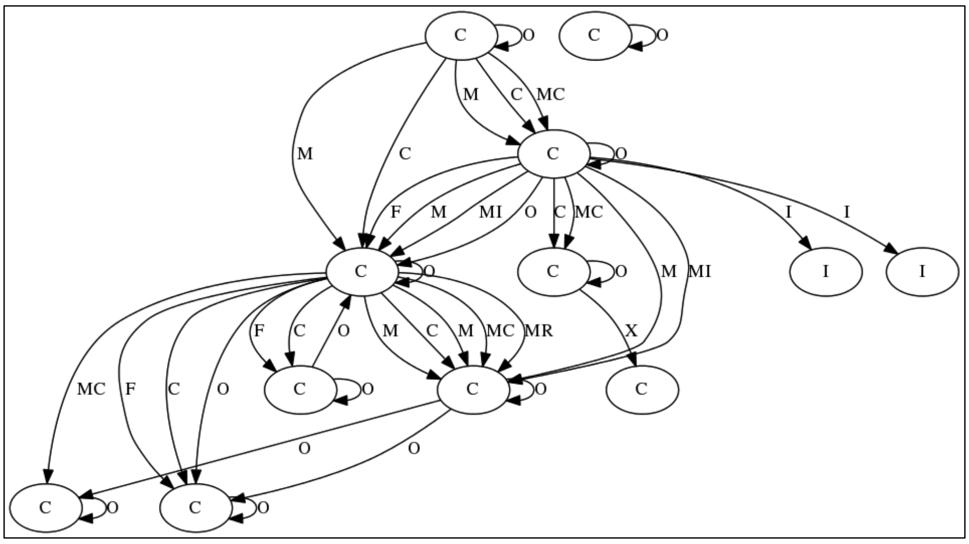


Figure 3.35. Graph Model of AJP - Facade example source code

As a result of the search, the detected pattern of Facade is demonstrated in Figure 3.36.

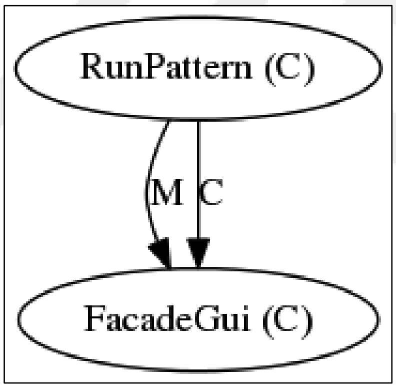


Figure 3.36. Facade Pattern detected in AJP

Flyweight: According to the literature this pattern is described as “Flyweight uses sharing to support large numbers of fine-grained objects efficiently” [4].

The Flyweight pattern’s influence relies upon on where and how it’s used. To implement this design pattern, a huge number of objects should be used in a software project. When the storage costs of software is high due to abrupt quantity of objects, then it is time for using Flyweight. Moreover, object identity is ignored in this pattern because the objects can be shared.

Figure 3.37 shows the template of Flyweight design pattern, which is defined in DesPaD.

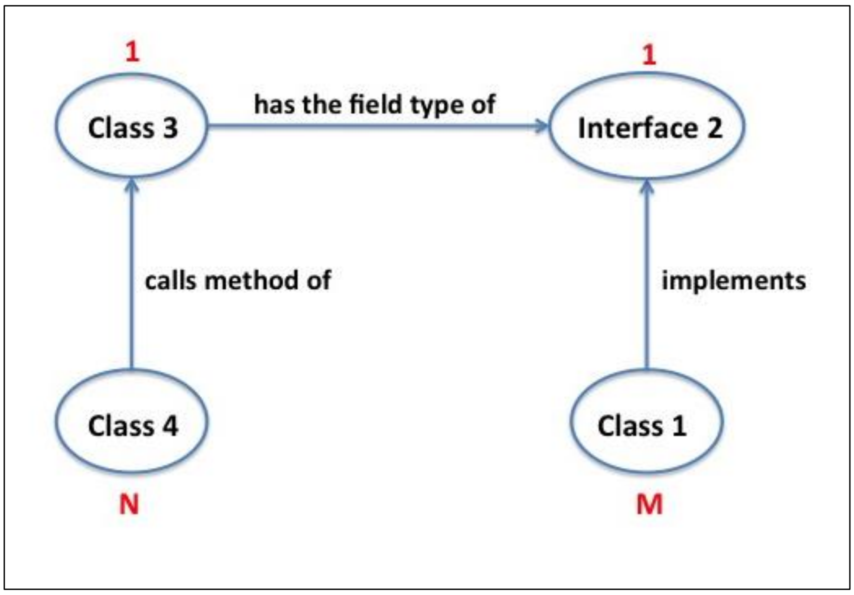


Figure 3.37. Template of Flyweight Design Pattern

Figure 3.38 shows the created model graph in the example source code in Applied Java Patterns book. The template of Flyweight Design Pattern is sought in this model graph.

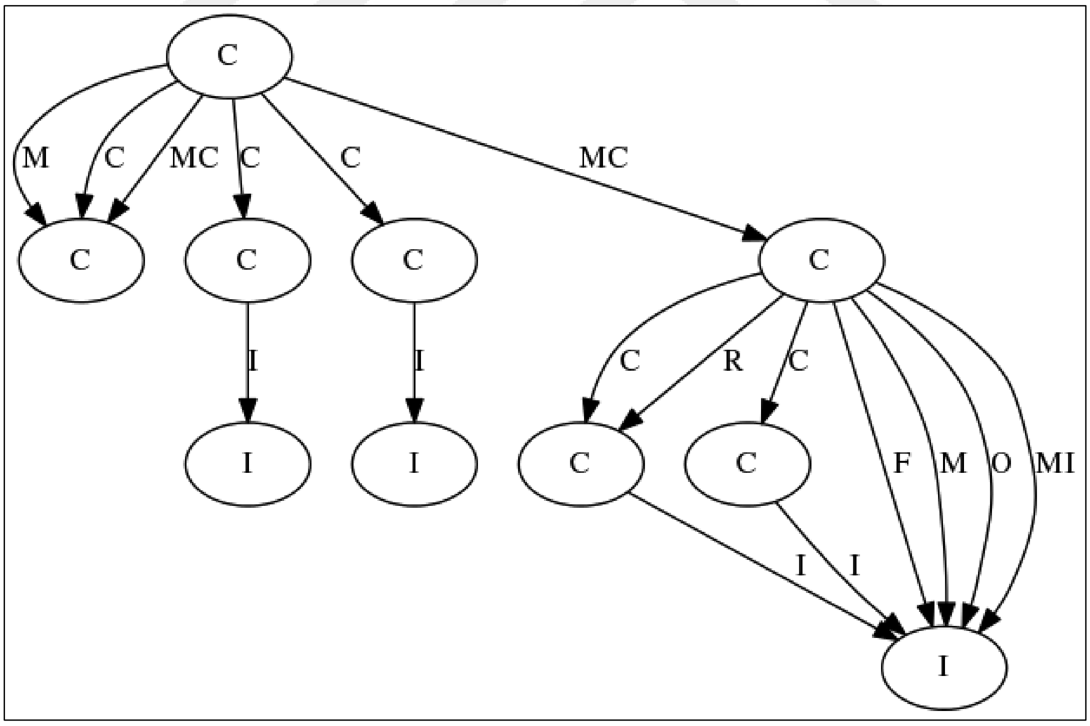


Figure 3.38. Graph Model of AJP - Flyweight example source code

As a result of the search, the detected pattern of Flyweight is demonstrated in Figure 3.39.

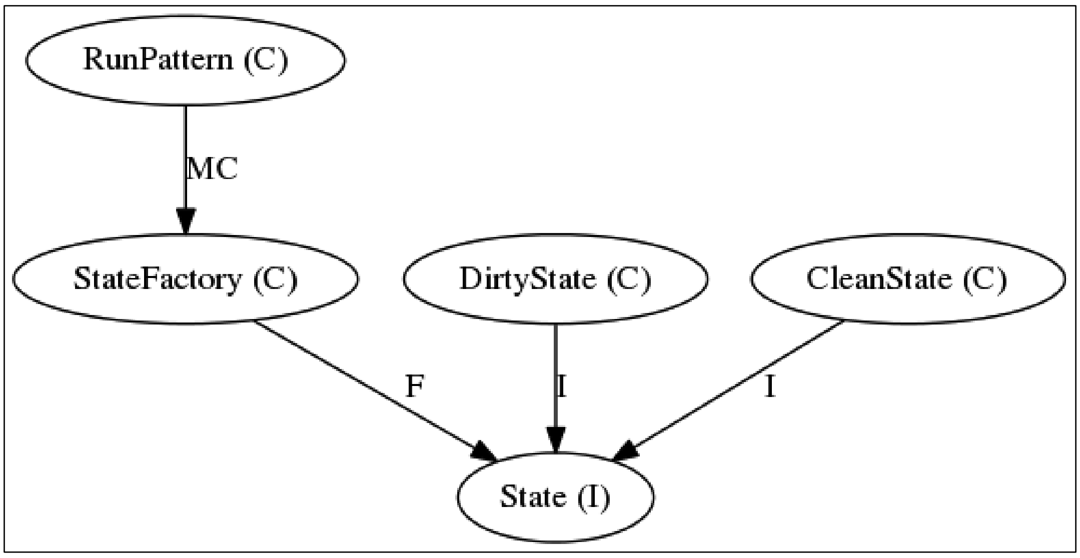


Figure 3.39. Flyweight Pattern detected in AJP

Proxy: According to the literature this pattern is described as “Proxy sustains a surrogate or placeholder for another object for controlling access to it” [4].

Proxy is executable when a developer needs a more flexible/sophisticated reference to an object than a simple pointer. Proxy pattern is applicable in many common situations. For example, a remote proxy supports a local representative for an object in a different address space. NXProxy is used for this goal in NEXTSTEP [42]. Coplien [41] uses “Ambassador” as a proxy. Moreover, a virtual proxy creates costly objects on demand. A projection proxy may be used if objects ought to have various access rights, since this proxy is able to regulate access to the original object. For instance, “KernelProxies in the Choices operating system” [43] support protected access to operating system objects. Furthermore, if an object is contacted, additional actions may be carried out by a substitute for a pointer named as smart reference.

Figure 3.40 shows the template of Proxy design pattern, which is defined in DesPaD.

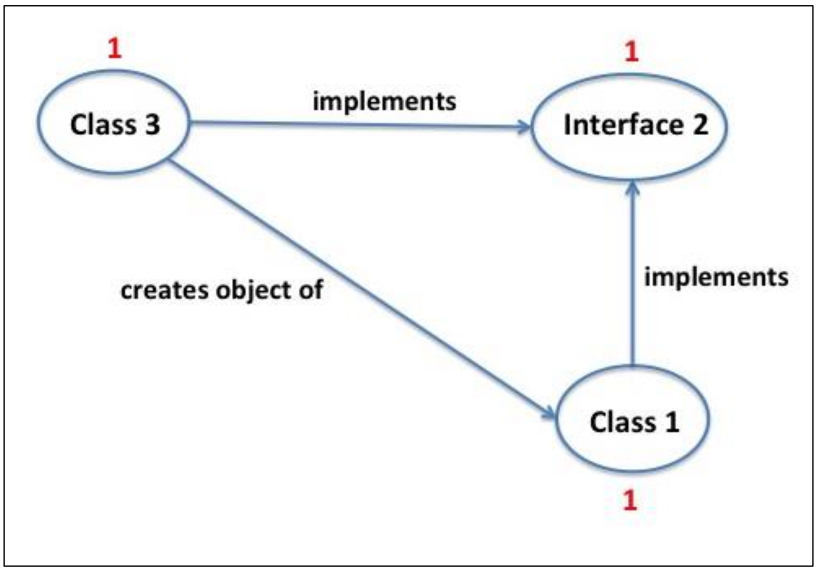


Figure 3.40. Template of Proxy Design Pattern

Figure 3.41 shows the created model graph in the example source code in Applied Java Patterns book. The template of Proxy Design Pattern is sought in this model graph.

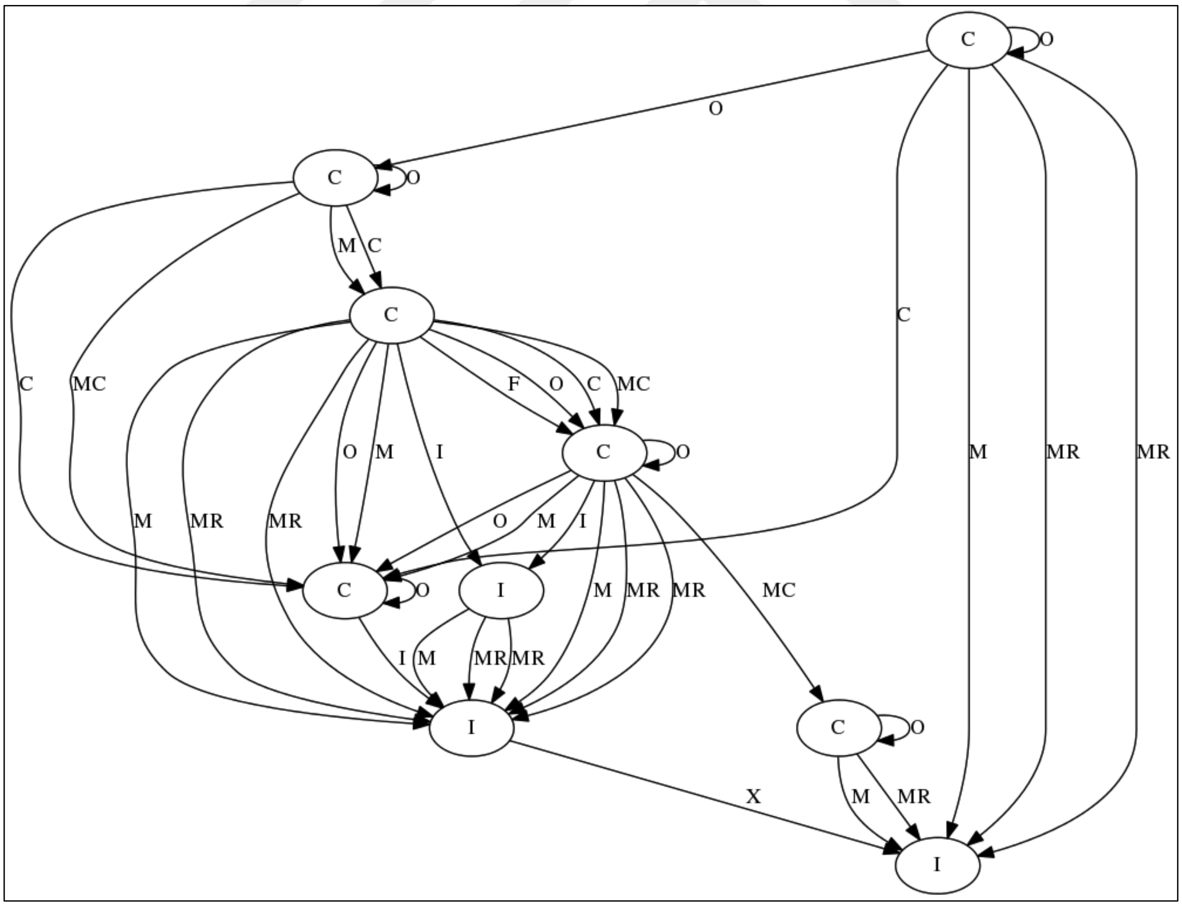


Figure 3.41. Graph Model of AJP - Proxy example source code

As a result of the search, the detected pattern of Proxy is demonstrated in Figure 3.42.

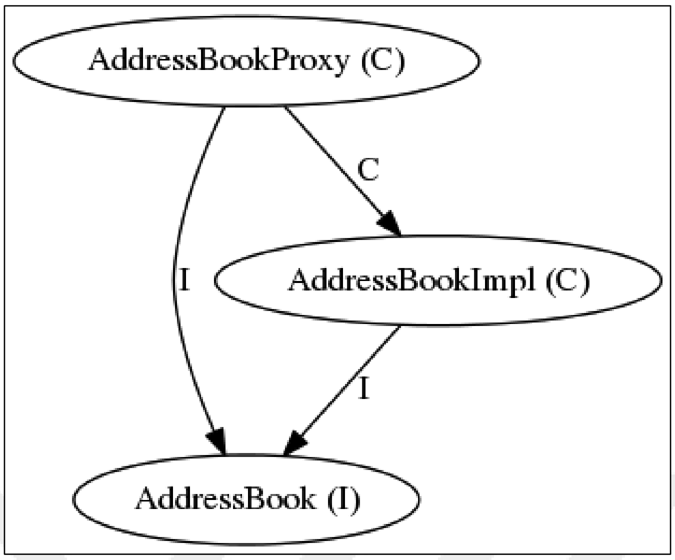


Figure 3.42. Proxy Pattern detected in AJP

3.4.3. Behavioral Design Patterns

In the literature, it is stated as “Behavioral patterns identify the ways where classes or objects interact and distribute responsibility” [4]. This class of patterns includes chain of responsibility, command, interpreter, iterator, mediator, memento, observer, state, strategy, template method and visitor.

Chain of Responsibility (CoR): According to the literature this pattern is described as “Chain of responsibility prevents coupling the sender of a request to its receiver by giving more than one object a chance to deal with the request and it chains the receiving objects and transmits the request along the chain until an object handles it” [4].

When a request may be handled by more than one object, and the handler isn’t known a priori, Chain of Responsibility design pattern is used. In this pattern, the handler should be determined automatically. CoR is useful while a developer wants to handle a request to one of various objects beyond designating the receiver explicitly.

Figure 3.43 shows the template of Chain of Responsibility design pattern, which is defined in DesPaD.

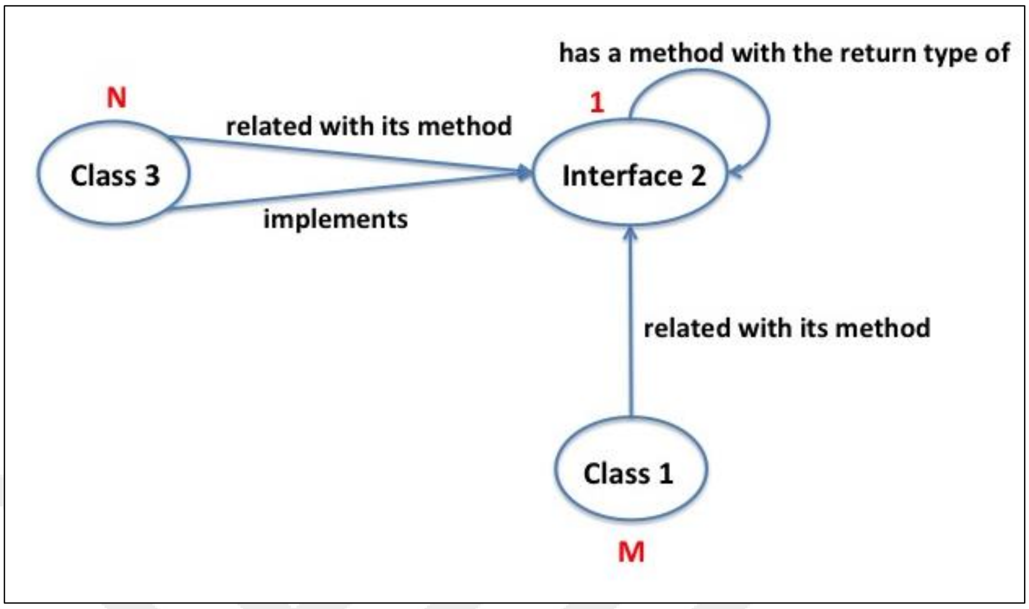


Figure 3.43. Template of Chain of Responsibility Design Pattern

Figure 3.44 shows the created model graph in the example source code in Applied Java Patterns book. The template of Chain of Responsibility Design Pattern is sought in this model graph.

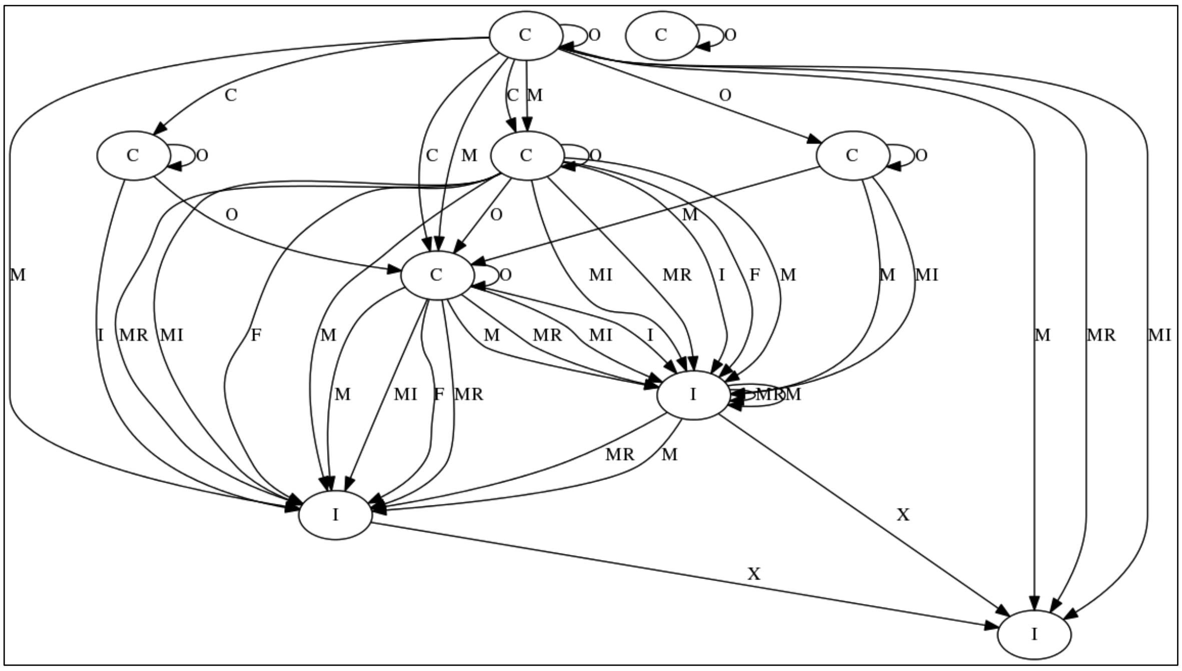


Figure 3.44. Graph Model of AJP - CoR example source code

As a result of the search, the detected pattern of Chain of Responsibility is demonstrated in Figure 3.45.

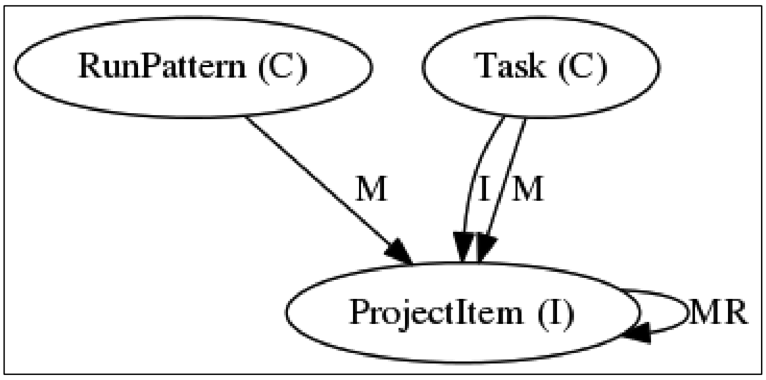


Figure 3.45. Chain of Responsibility Pattern detected in AJP

Command: According to the literature this pattern is described as “Command encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and assist undoable operations” [4].

The requirement of parameterizing objects by a method operation is accomplished by Command pattern. Such parameterization is similar to a callback method in a procedural language. Callback is a registration of a point to be called later. Command pattern is an alternative for those callback methods in an object-oriented programming language. Moreover, this pattern becomes useful when one wants to send requests that may be defined, queued, and then executed at particular times. Non-dependent to the original request, an object of Command is able to have a lifetime.

The Command’s Execute operation can store state; by this it can rollback itself. Executed commands are saved in a history list. Additionally, this pattern supports logging modifications, so in case of system failure those log changes can be re-executed.

Figure 3.46 shows the template of Command design pattern, which is defined in DesPaD.

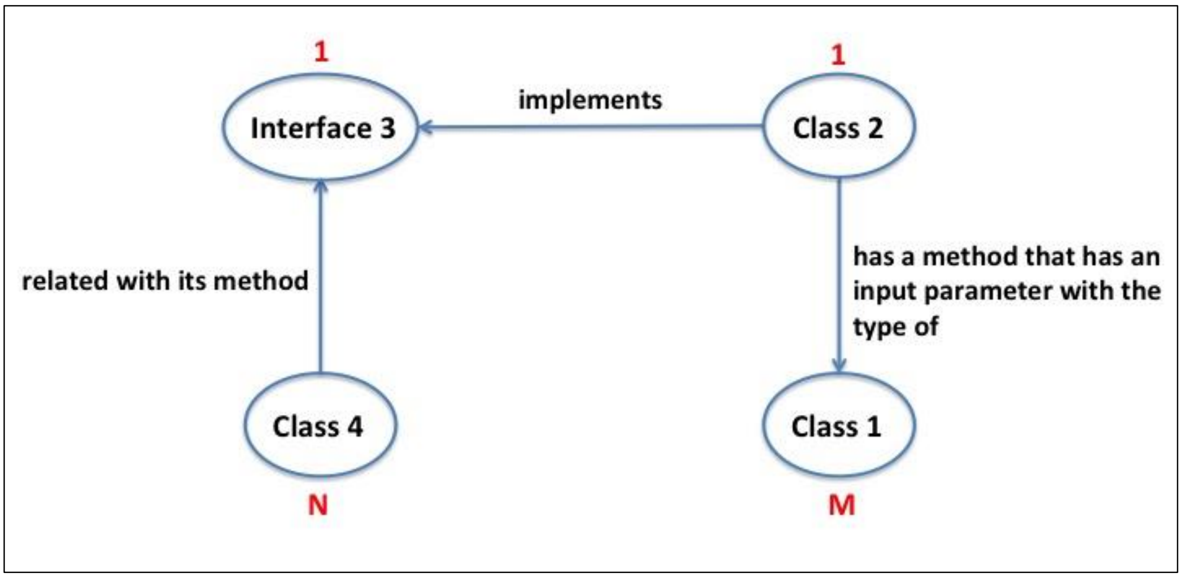


Figure 3.46. Template of Command Design Pattern

Figure 3.47 shows the created model graph in the example source code in Applied Java Patterns book. The template of Command Design Pattern is sought in this model graph.

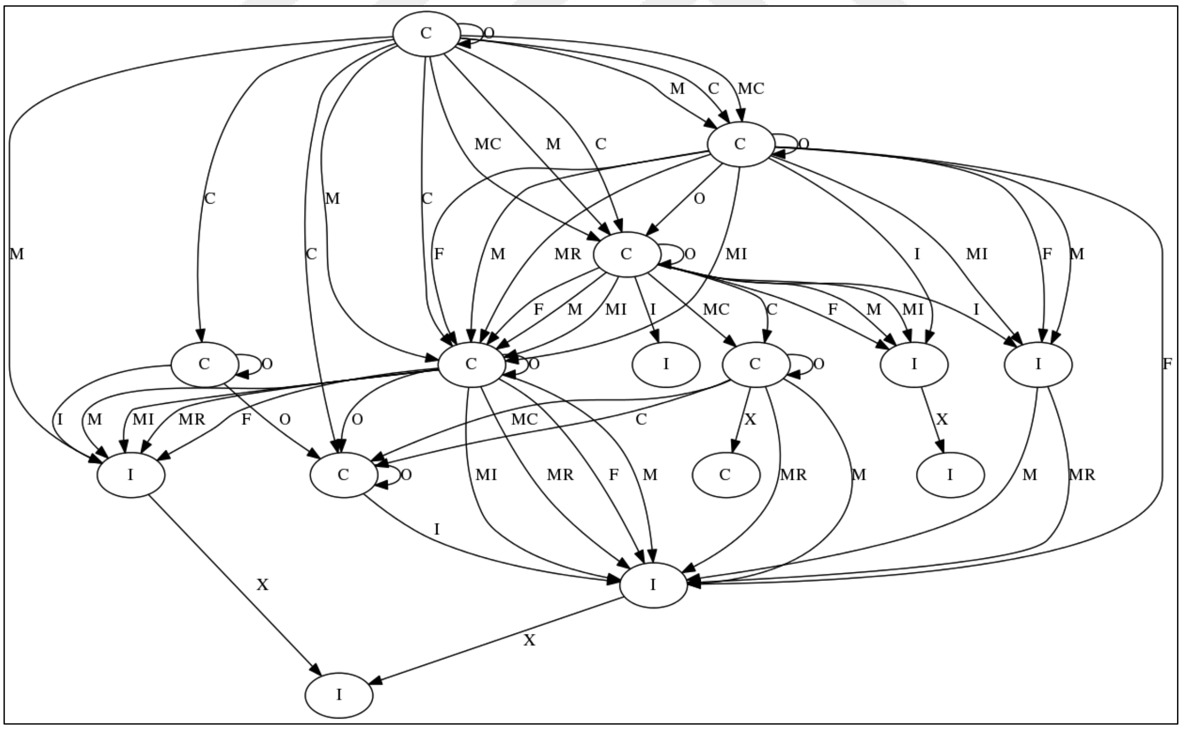


Figure 3.47. Graph Model of AJP - Command example source code

As a result of the search, the detected pattern of Command is demonstrated in Figure 3.48.

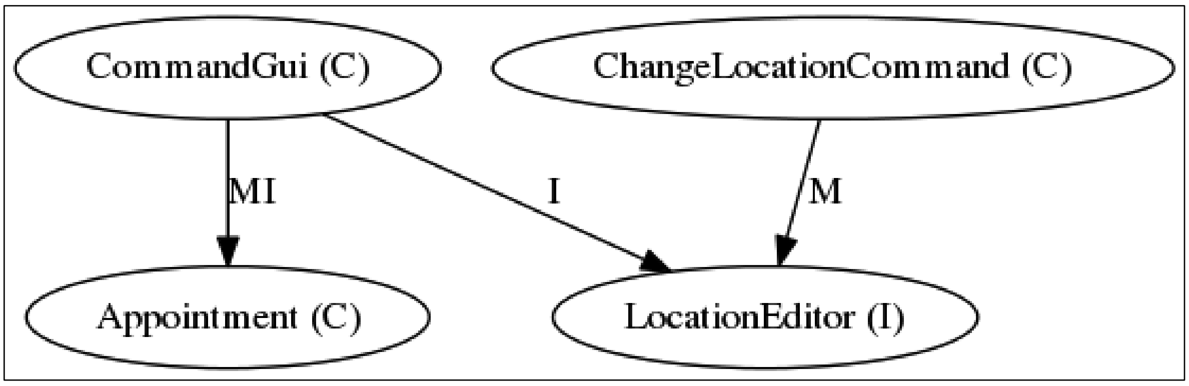


Figure 3.48. Command Pattern detected in AJP

Interpreter: According to the literature this pattern is described as "When given a language, Interpreter design pattern outlines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language" [4].

Interpreter pattern is used for interpreting a language, and one can symbolize language statements as abstract syntax trees (AST).

If the grammar of a language is simple, then Interpreter pattern may be used. The class hierarchies for the complex grammars become large and uncontrolled. In these kinds of cases, some tools like parser generators are a smarter solution. They are able to interpret expressions without building ASTs, which is less resource-consuming. On the other hand, one cannot expect efficiency from this pattern. The most efficient interpreters are usually translating parse trees into another form instead of implementing the parse trees directly. Regular expressions are a good example for this. They are frequently transformed into state machines.

Figure 3.49 shows the template of Interpreter design pattern, which is defined in DesPaD.

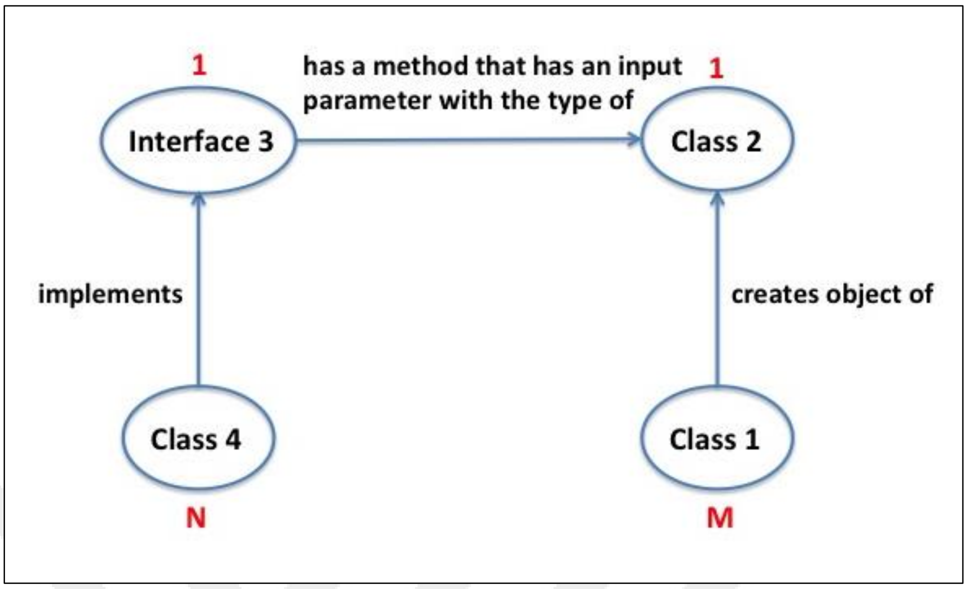


Figure 3.49. Template of Interpreter Design Pattern

Figure 3.50 shows the created model graph in the example source code in Applied Java Patterns book. The template of Interpreter Design Pattern is sought in this model graph.

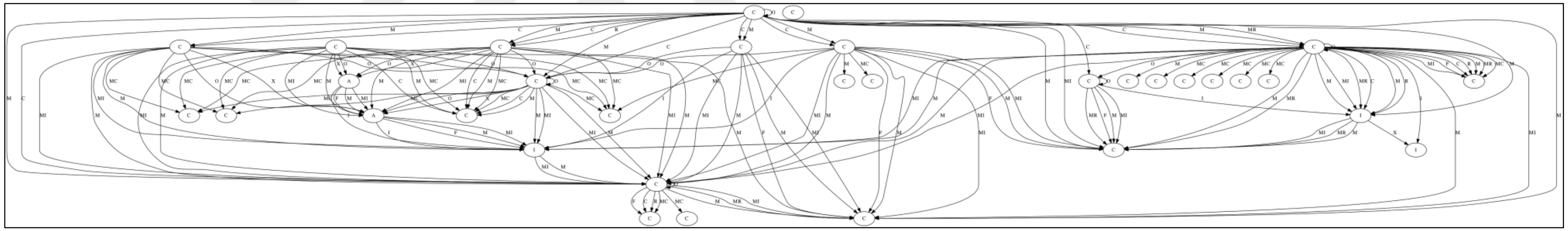


Figure 3.50. Graph Model of AJP - Interpreter example source code

As a result of the search, the detected pattern of Interpreter is demonstrated in Figure 3.51.

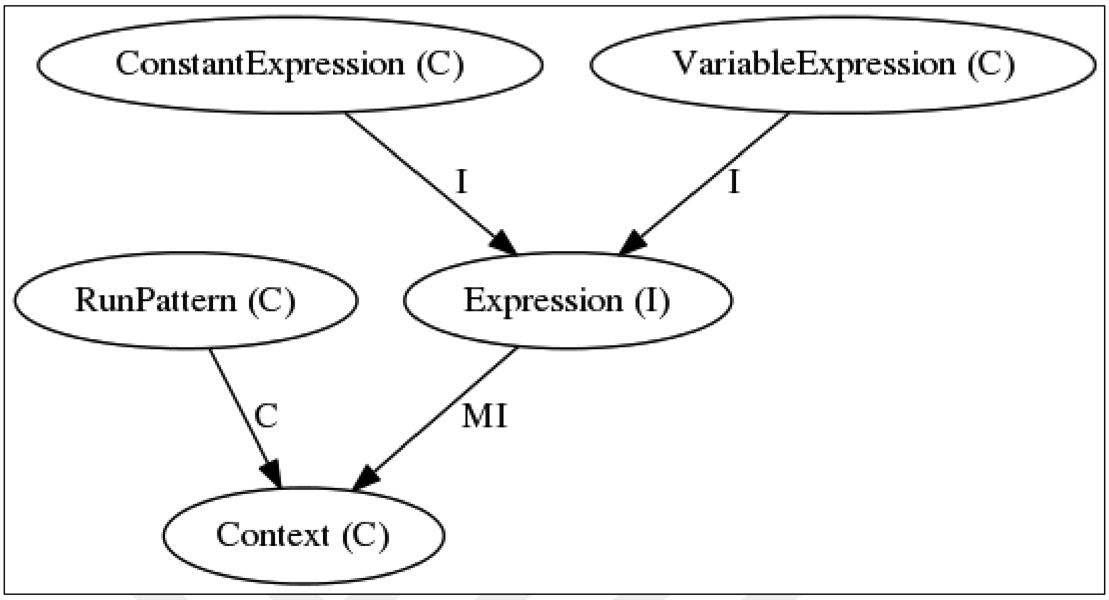


Figure 3.51. Interpreter Pattern detected in AJP

Iterator: According to the literature this pattern is described as “Iterator supports a way to access the items of an aggregate object sequentially without exposing its underlying representation” [4].

When a developer wants to reach an aggregate object’s contents, but one does not want to reveal its internal representation, Iterator is the solution. It also provides numerous traversals of aggregate objects. Moreover, this pattern supports a uniform interface for walking on various aggregate structures.

Figure 3.52 shows the template of Iterator design pattern, which is defined in DesPaD.

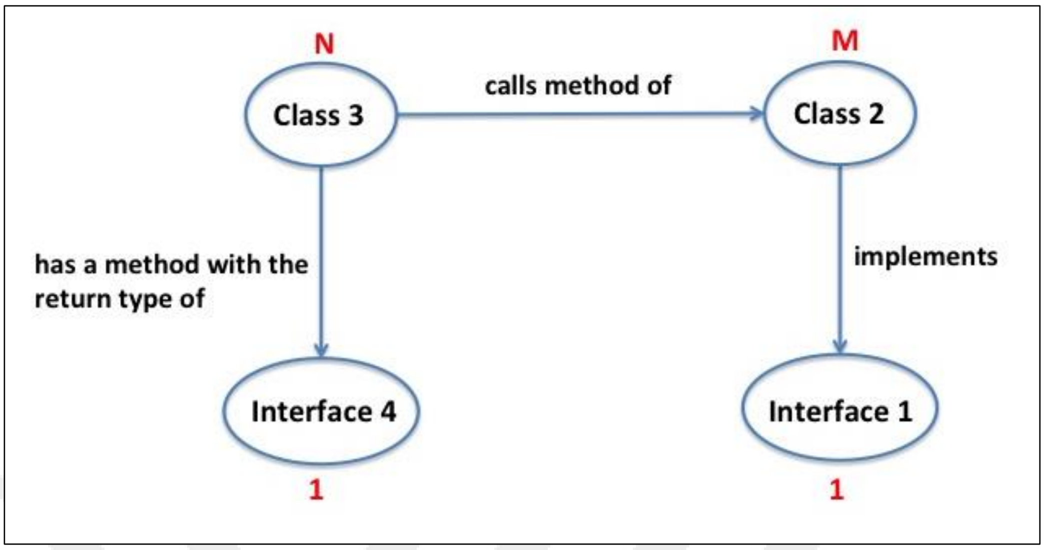


Figure 3.52. Template of Iterator Design Pattern

Figure 3.53 shows the created model graph in the example source code in Applied Java Patterns book. The template of Iterator Design Pattern is sought in this model graph.

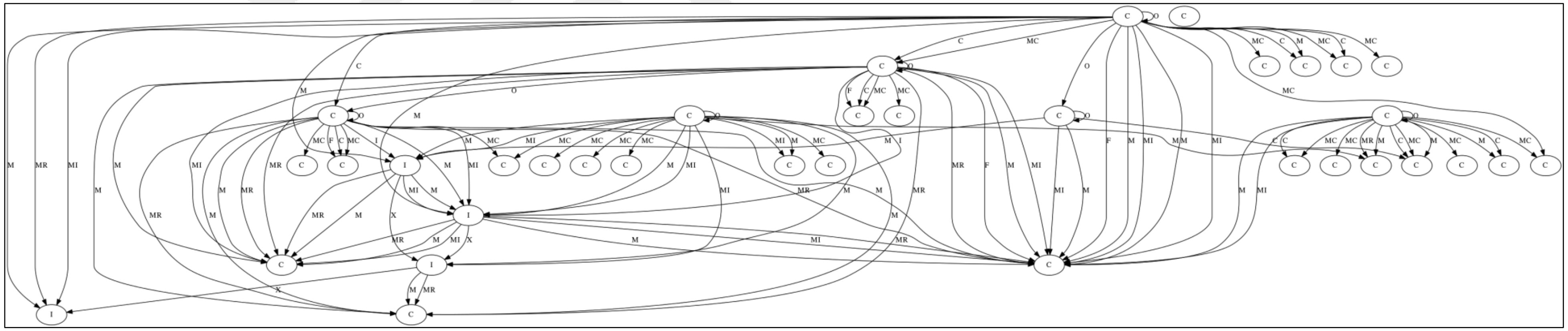


Figure 3.53. Graph Model of AJP - Iterator example source code

As a result of the search, the detected pattern of Iterator is demonstrated in Figure 3.54.

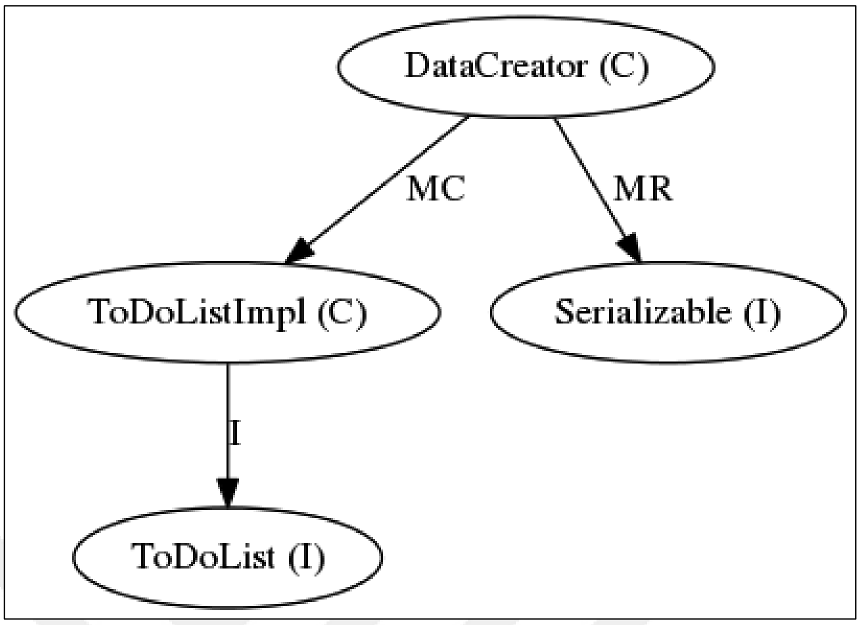


Figure 3.54. Iterator Pattern detected in AJP

Mediator: According to the literature this pattern is described as “Mediator specifies an object that encapsulates how a set of objects interacts and also promotes loose coupling by holding objects from referring to each other explicitly, and lets you change their interaction independently” [4].

In this pattern, a set of objects communicates in well-defined but complicated ways. The dependencies between objects are unstructured and hard to figure out. Mediator design pattern gives references to and communicates with many other objects, because of that reusing an object is challenging. Its behavior which is distributed between many classes may be customizable without a plenty of subclassing.

Figure 3.55 shows the template of Mediator design pattern, which is defined in DesPaD.

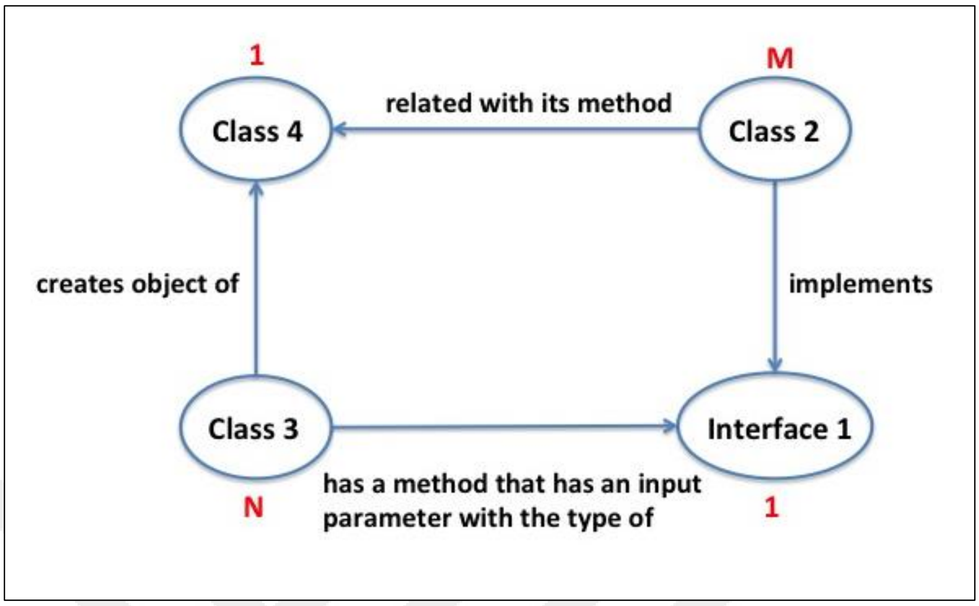


Figure 3.55. Template of Mediator Design Pattern

Figure 3.56 shows the created model graph in the example source code in Applied Java Patterns book. The template of Mediator Design Pattern is sought in this model graph.

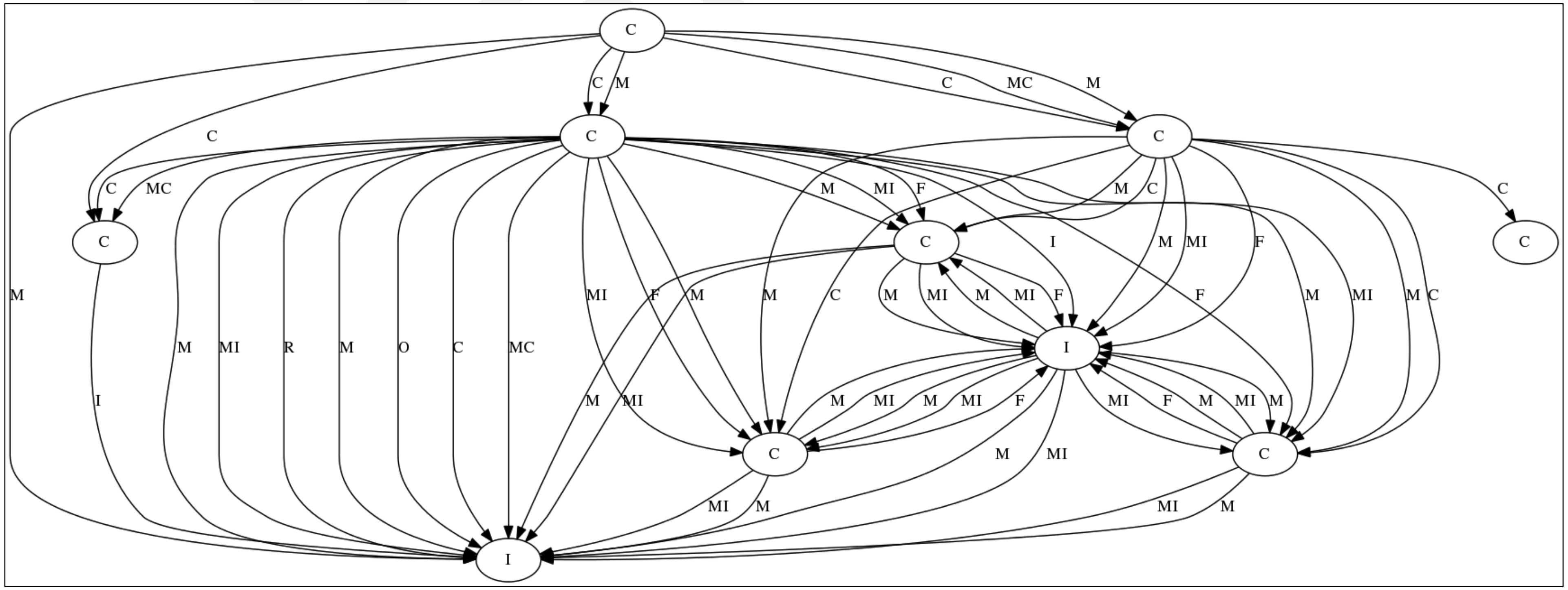


Figure 3.56. Graph Model of AJP - Mediator example source code

As a result of the search, the detected pattern of Mediator is demonstrated in Figure 3.57.

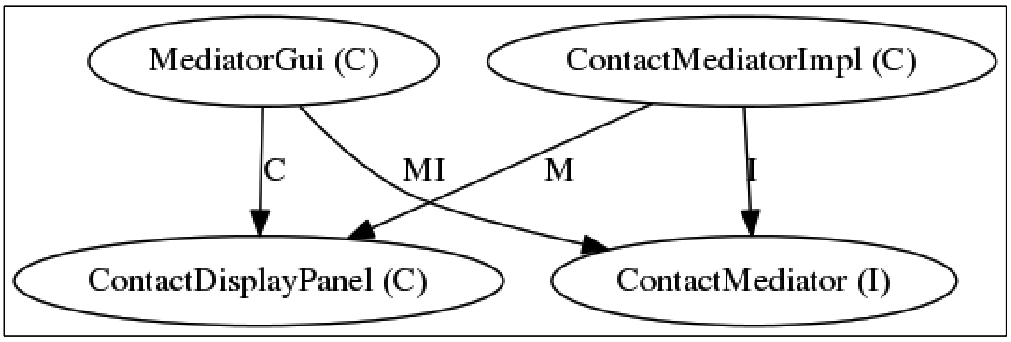


Figure 3.57. Mediator Pattern detected in AJP

Memento: According to the literature this pattern is described as “Without disobeying encapsulation, Memento pattern captures and externalizes an object’s internal state so that the object can be restored to this state afterwards” [4].

Memento may be preferred if a snapshot of an object’s state should be stored, then it may be re-stored to that state afterwards, and a direct interface to getting the state may reveal implementation details and break the object’s encapsulation.

Figure 3.58 shows the template of Memento design pattern, which is defined in DesPaD.

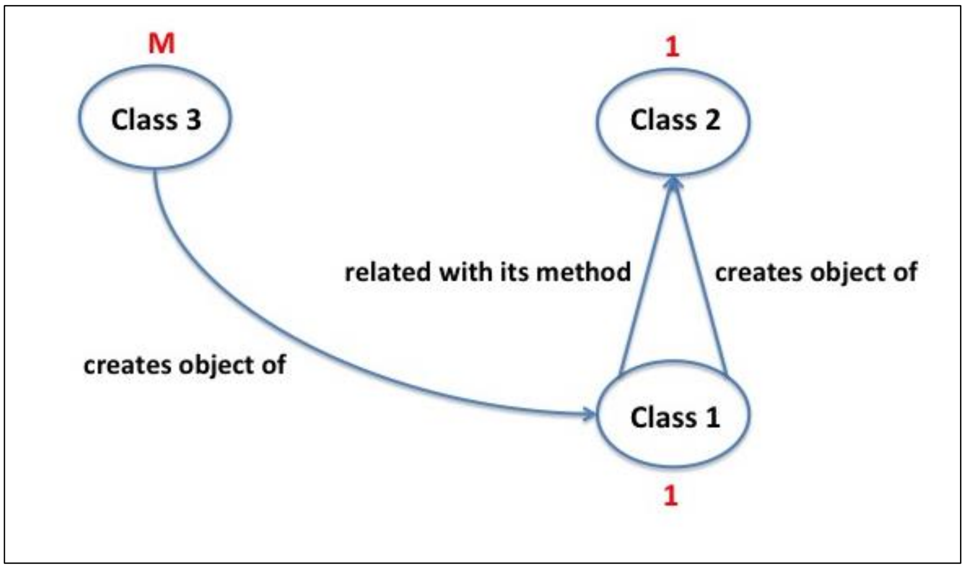


Figure 3.58. Template of Memento Design Pattern

Figure 3.59 shows the created model graph in the example source code in Applied Java Patterns book. The template of Memento Design Pattern is sought in this model graph.

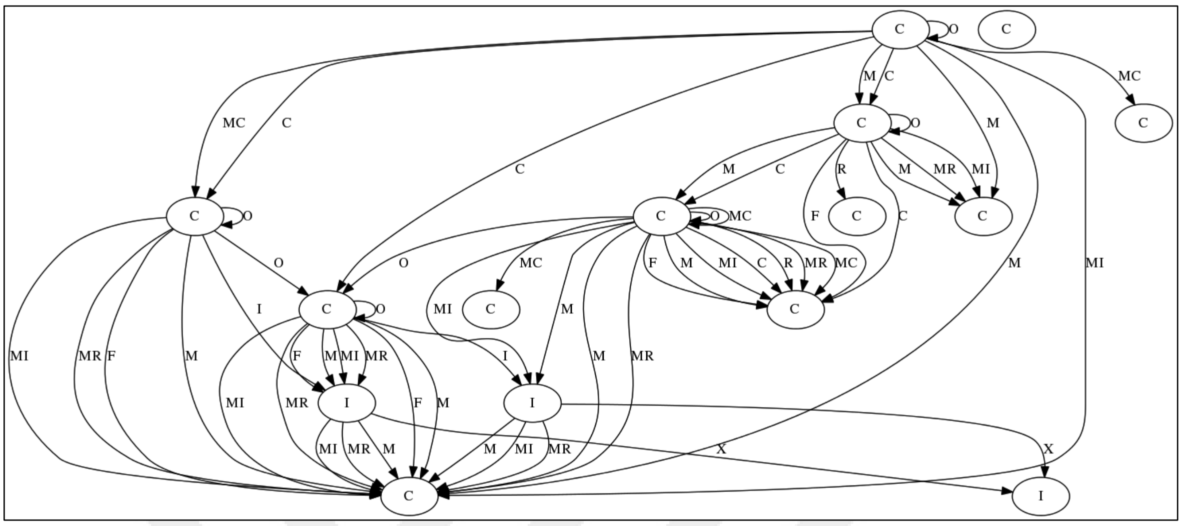


Figure 3.59. Graph Model of AJP - Memento example source code

As a result of the search, the detected pattern of Memento is demonstrated in Figure 3.60.

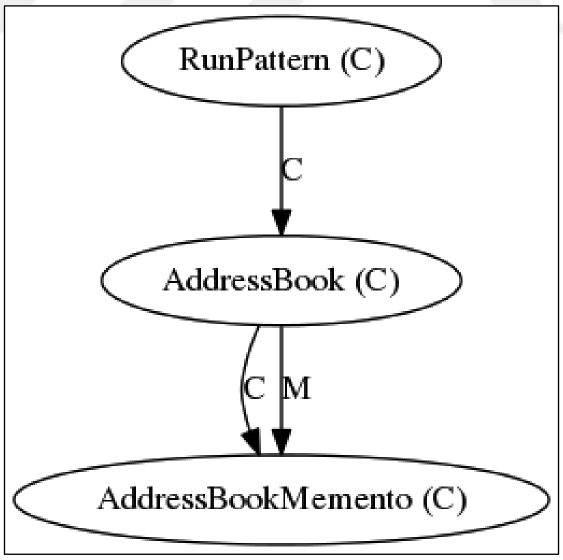


Figure 3.60. Memento Pattern detected in AJP

Observer: According to the literature this pattern is described as "Observer explains a one-to-many dependency between objects so that when one object alters state, all its dependents are informed and updated in an automated-manner" [4].

Observer pattern may be used when an abstraction has two facets, which are dependent on each other. If one wants to encapsulate these aspects in separate objects, the use of observer pattern may be used to modify and reuse them non-dependently. The pattern may be preferred also when one object's modification needs modifying others, and the exact number of objects require to be changed are not known. If an object needs be able to give notification to other objects without making assumptions on recognizing them, the developer may choose to apply the Observer pattern.

Figure 3.61 shows the template of Observer design pattern, which is defined in DesPaD.

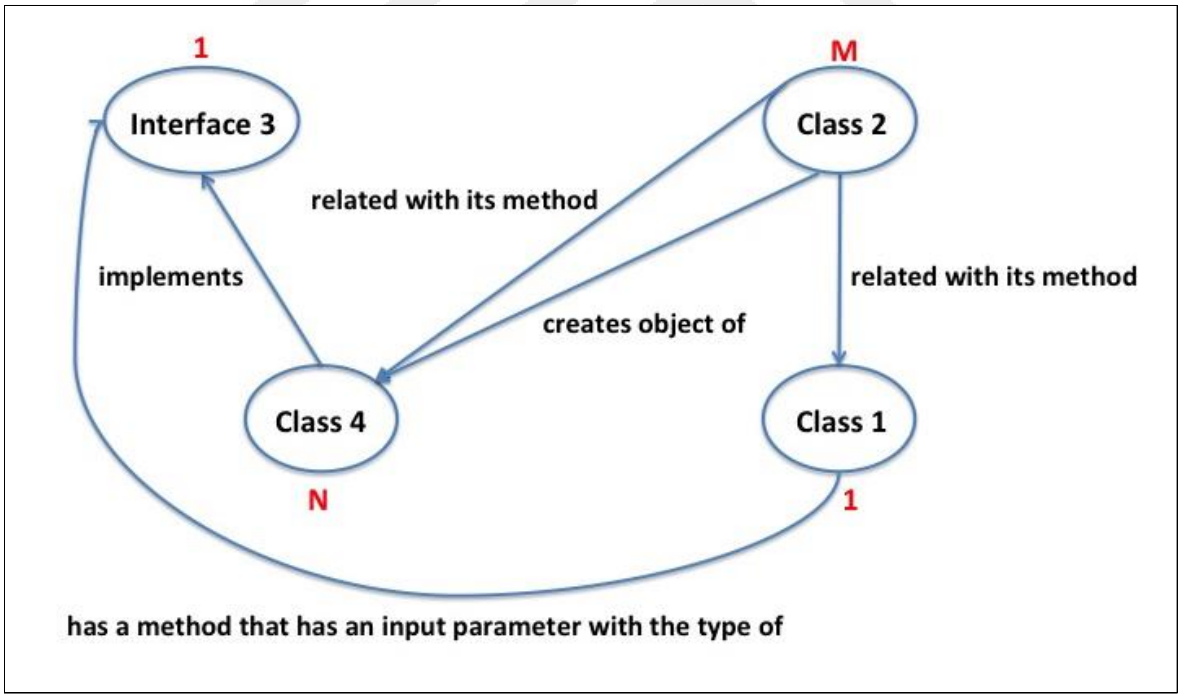


Figure 3.61. Template of Observer Design Pattern

Figure 3.62 shows the created model graph in the example source code in Applied Java Patterns book. The template of Observer Design Pattern is sought in this model graph.

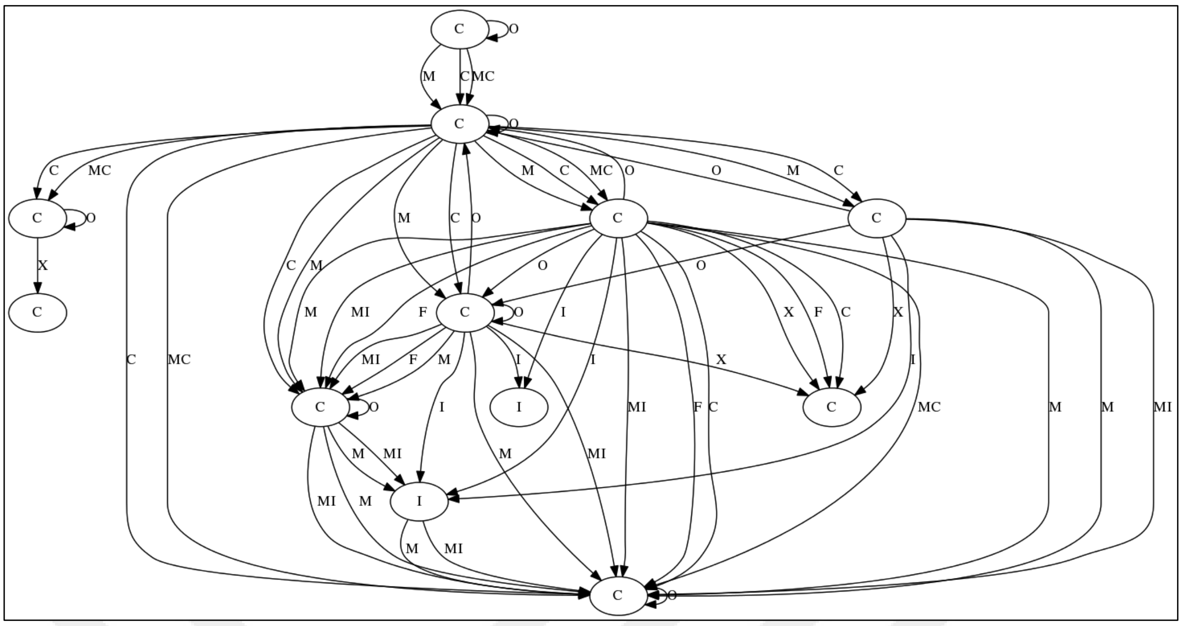


Figure 3.62. Graph Model of AJP - Observer example source code

As a result of the search, the detected pattern of Observer is demonstrated in Figure 3.63.

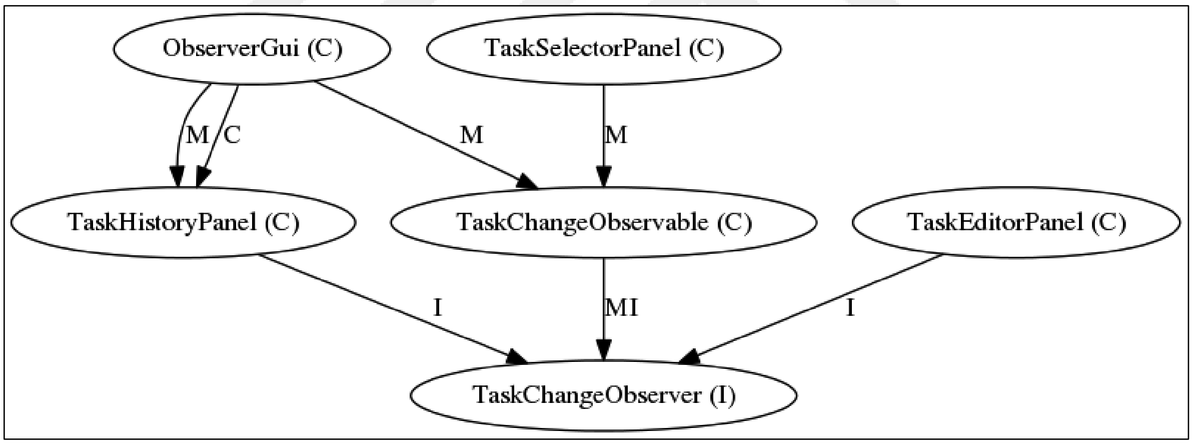


Figure 3.63. Observer Pattern detected in AJP

State: According to the literature this pattern is described as “State lets an object change its behavior when its internal state alters. The object will come out to change its class” [4].

When an object’s behavior changes according to its state, and its behavior should be modified at run-time based on that state, State pattern’s usage is possible. In this pattern, depending on the object’s state, operations have large, multiple conditional statements. This state is usually served by one or several enumerated constants. Frequently, some operations will include this same conditional structure.

Figure 3.64 shows the template of State design pattern, which is defined in DesPaD.

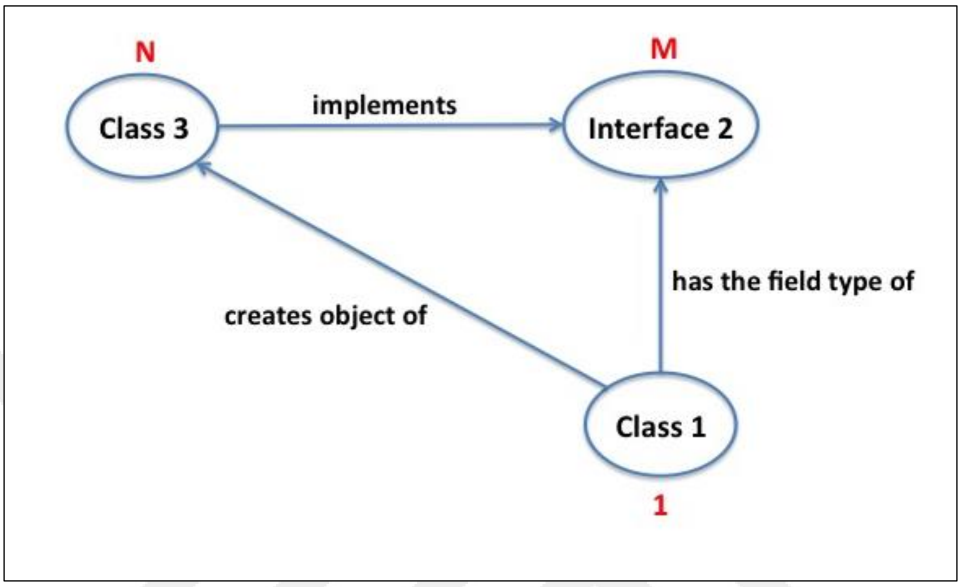


Figure 3.64. Template of State Design Pattern

Figure 3.65 shows the created model graph in the example source code in Applied Java Patterns book. The template of State Design Pattern is sought in this model graph.

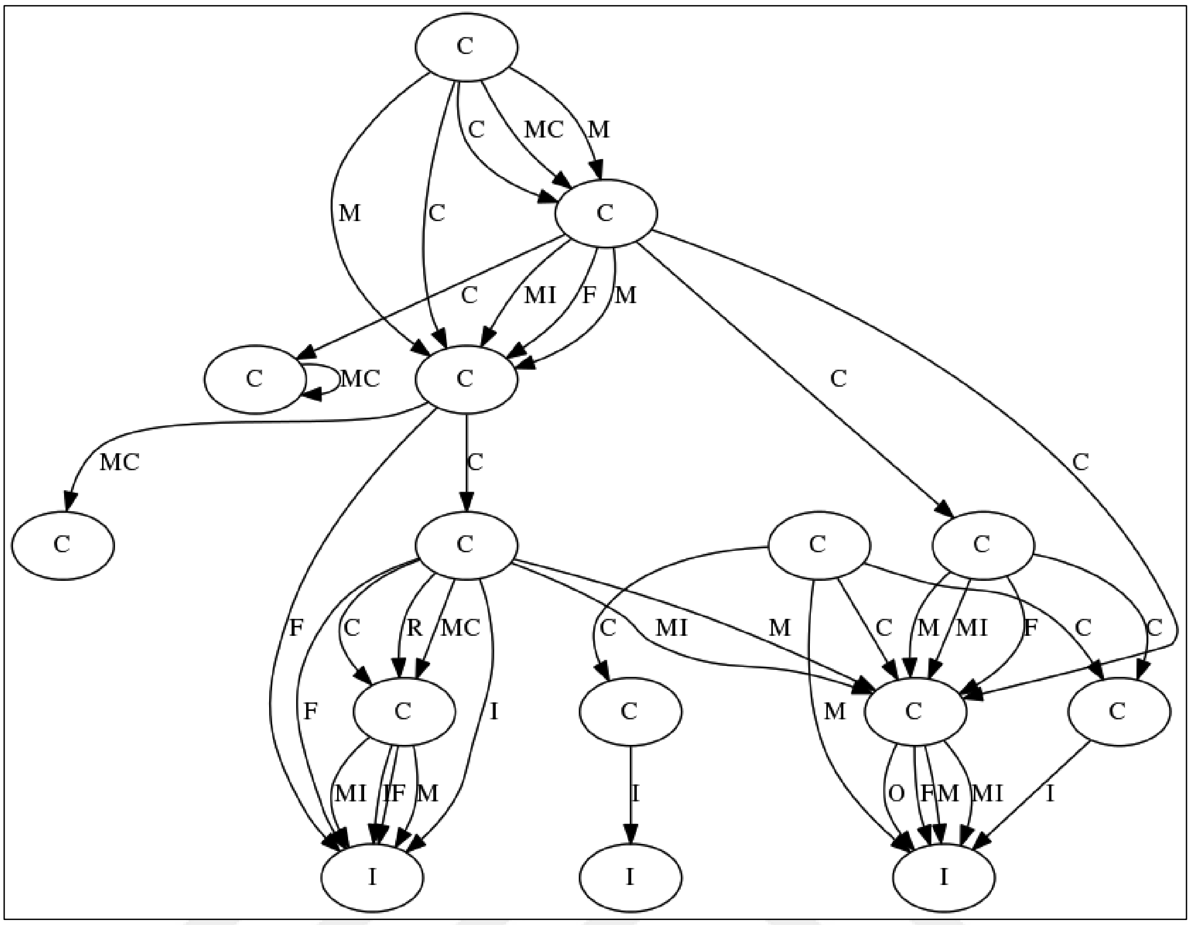


Figure 3.65. Graph Model of AJP - State example source code

As a result of the search, the detected pattern of State is demonstrated in Figure 3.66.

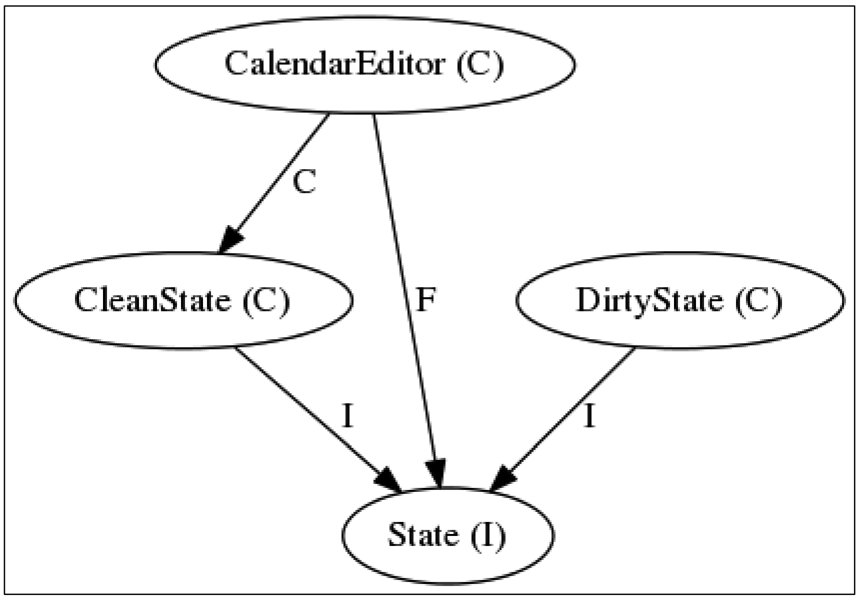


Figure 3.66. State Pattern detected in AJP

Strategy: According to the literature this pattern is described as “Strategy defines a family of algorithm, encapsulate each one, and make them interchangeable and allows the algorithm to change independently from clients that use it” [4].

Sometimes a developer or an architect of a software project needs different variants of an algorithm; in this case Strategy pattern is the solution. For instance, one may characterize algorithms that reflect different resource or time tradeoffs. These variants in Strategy pattern are executed as a class hierarchy of algorithms [44]. An algorithm benefits from data which clients shouldn’t know about.

Strategy design pattern is used to prevent exposing algorithm-specific and complex data structures. A class defines many behaviors, and these behaviors appear as multiple conditional statements in its operations. Instead of this, one may move related conditional branches into their own Strategy class.

Figure 3.67 shows the template of Strategy design pattern, which is defined in DesPaD.

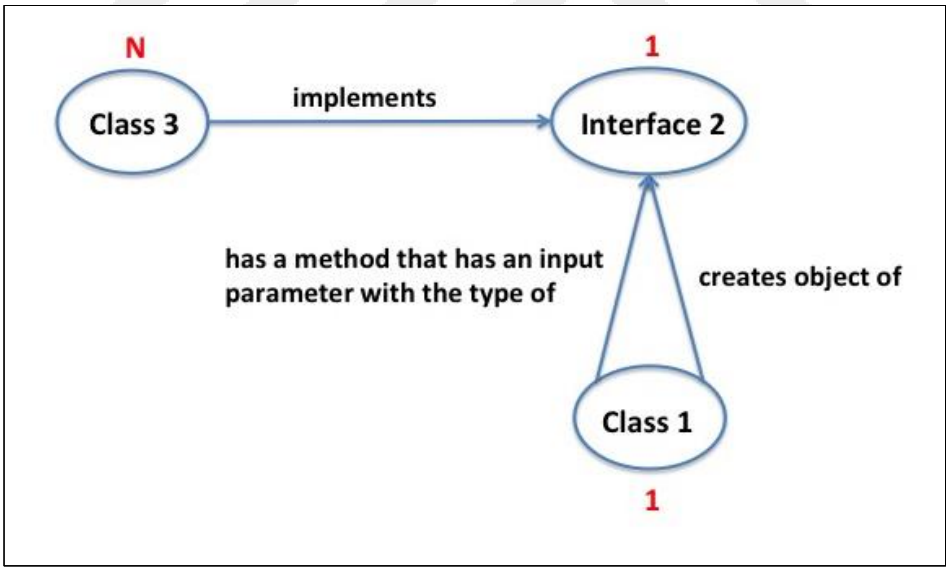


Figure 3.67. Template of Strategy Design Pattern

Figure 3.68 shows the created model graph in the example source code in Applied Java Patterns book. The template of Strategy Design Pattern is sought in this model graph.

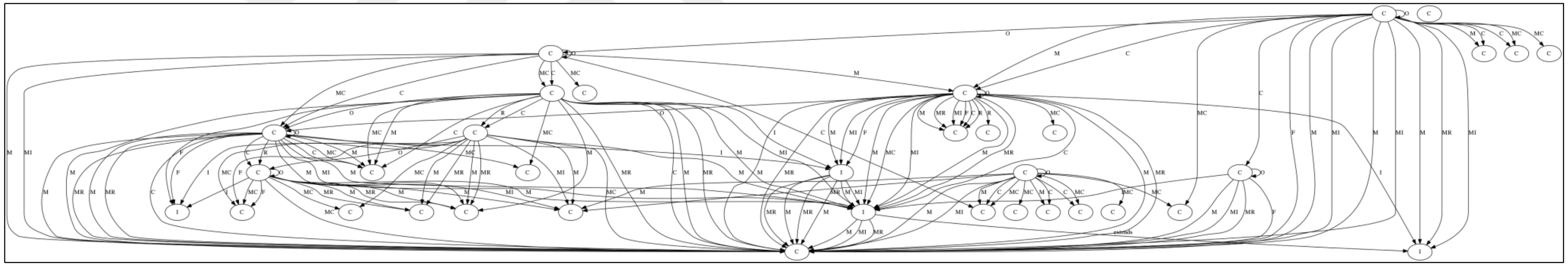


Figure 3.68. Graph Model of AJP - Strategy example source code

As a result of the search, the detected pattern of Strategy is demonstrated in Figure 3.69.

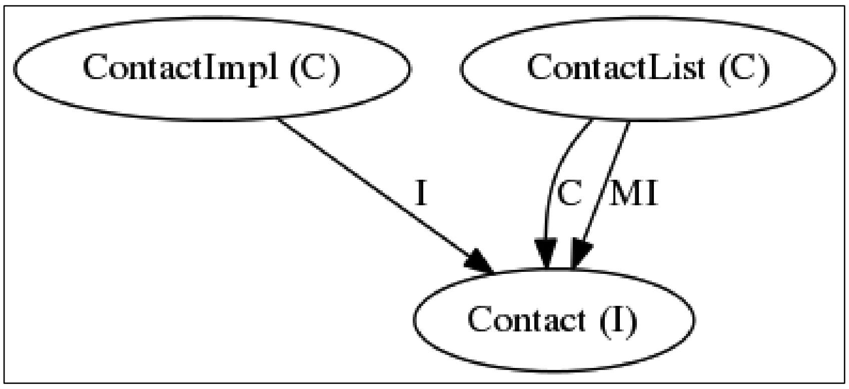


Figure 3.69. Strategy Pattern detected in AJP

Template Method: According to the literature this pattern is described as "Template Method gives description of the skeleton of an algorithm in an operation, deferring some steps to subclasses and allows subclasses to redefine certain steps of an algorithm without altering the structure of the algorithm" [4].

Template method may be used when a developer first wants to implement the invariant parts of an algorithm and then leave it up to subclasses to execute the behavior that can change. This pattern also prevents duplicate code by common behavior among subclasses that are factored and localized in a common class. In [45], there is a good example of "refactoring to generalize". The dissimilarities have to be found first in the source code and then the differences have to be separated into new operations. Finally, the related code with a template method which is the pattern calling one of these new operations has to be replaced.

Figure 3.70 and Figure 3.71 show the templates of Template Method design pattern, which are defined in DesPaD.

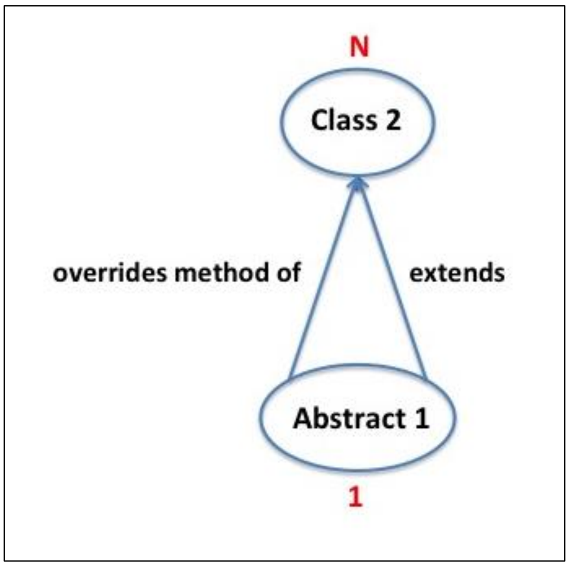


Figure 3.70. Template-1 of Template Method Design Pattern

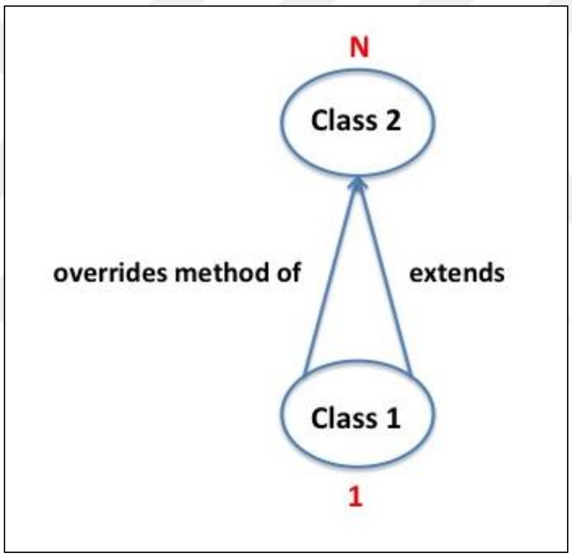


Figure 3.71. Template-2 of Template Method Design Pattern

Figure 3.72 shows the created model graph in the example source code in Applied Java Patterns book. The templates of Template Method Design Pattern are sought in this model graph.

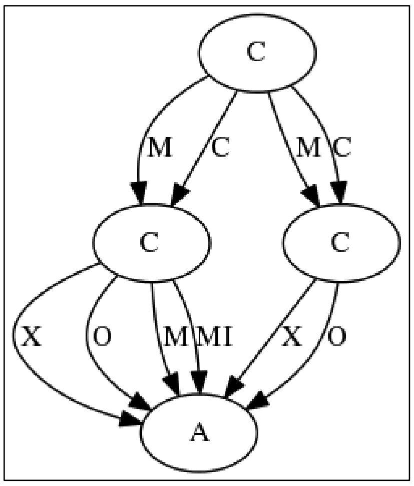


Figure 3.72. Graph Model of AJP – Template Method example source code

As a result of the search, the detected pattern of Template Method is demonstrated in Figure 3.73.

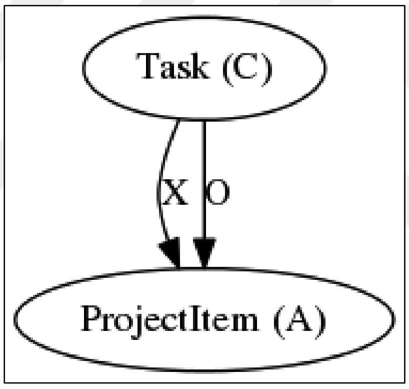


Figure 3.73. Template Method Pattern detected in AJP

Visitor: According to the literature this pattern is described as “Visitor exhibits an operation to be performed on the elements of an object structure and allows oneself to define a new operation without changing the classes of the elements on which it performs” [4].

Visitor pattern may be chosen as a solution if a developer wants an object structure consisting of many classes with different interfaces, and to execute operations on these objects according to their concrete classes. Moreover, the pattern may be good to use if the developer wants to prevent unrelated operations that are going to be performed on objects, which may be defined as the ones “polluting” their classes. With the help of this pattern, related operations may be kept together by defining all in one class.

The pattern may also be used in the case of classes that define occasionally changing object structures having new operations described over them. In order to change the object structure, a class needs re-defining the interface to all visitors, which may become potentially more expensive. If the object structure classes modify too often, giving description to the operations in those classes may become a wiser solution.

Figure 3.74 shows the template of Visitor design pattern, which is defined in DesPaD.

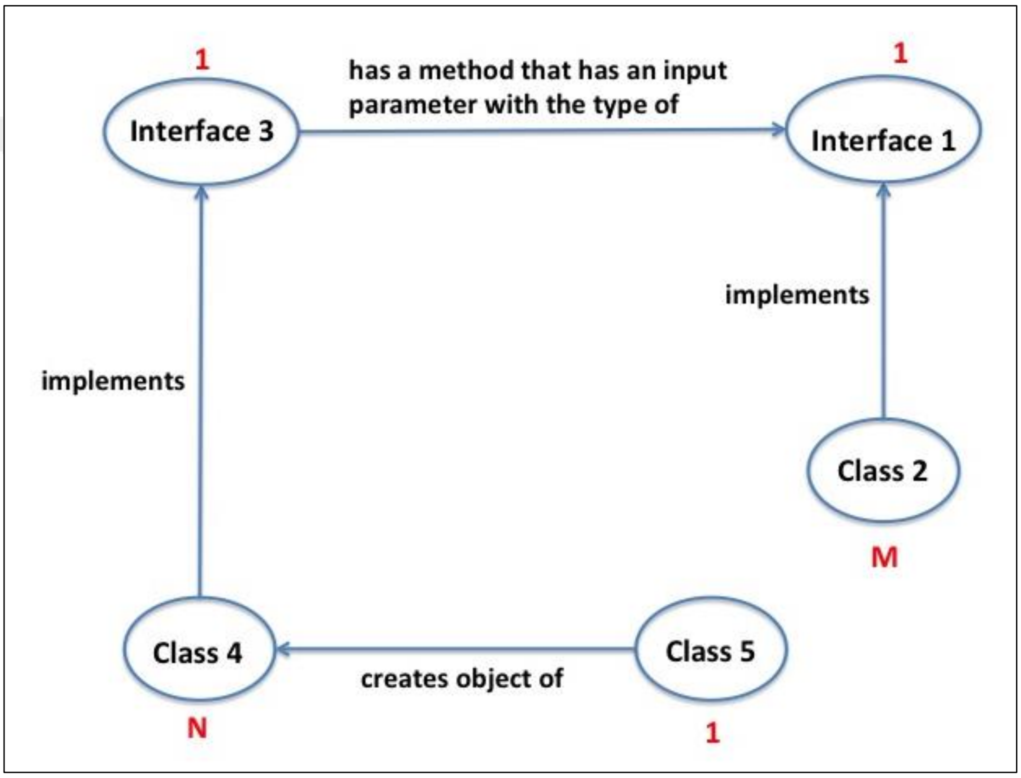


Figure 3.74. Template of Visitor Design Pattern

Figure 3.75 shows the created model graph in the example source code in Applied Java Patterns book. The template of Visitor Design Pattern is sought in this model graph.

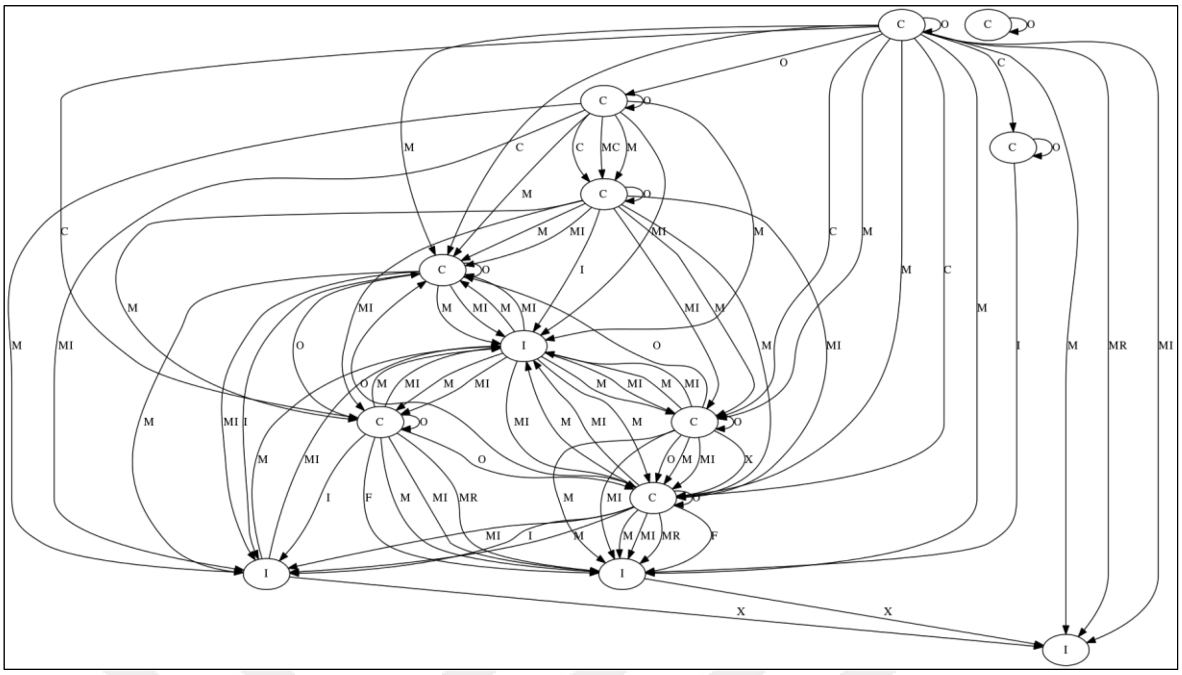


Figure 3.75. Graph Model of AJP – Visitor example source code

As a result of the search, the detected pattern of Visitor is demonstrated in Figure 3.76.

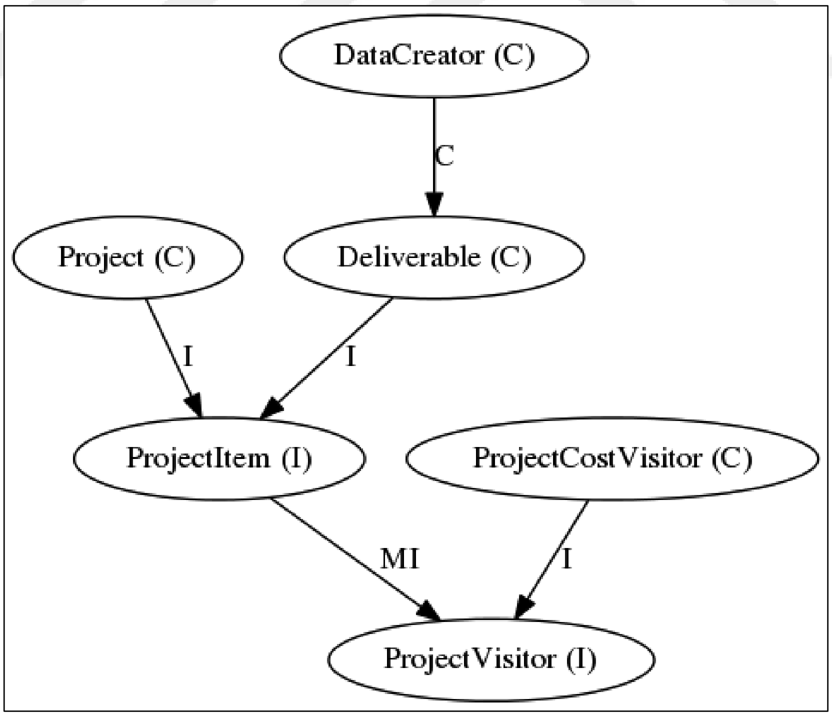


Figure 3.76. Visitor Pattern detected in AJP

4. DESIGN PATTERN DETECTOR TOOL (DesPaD)

4.1. Overview

A design pattern detection tool based on subgraph mining is implemented within this thesis. The tool is developed in Eclipse platform by using Java programming language. It is called DesPaD which stands for *Design Pattern Detector*. The tool provides a user-friendly interface seen in Figure 4.2. The user interface was developed by using JForm Designer plug-in for Java Swing user interfaces [36]. JFormDesigner [36] is for creating a user-friendly desktop application easily.

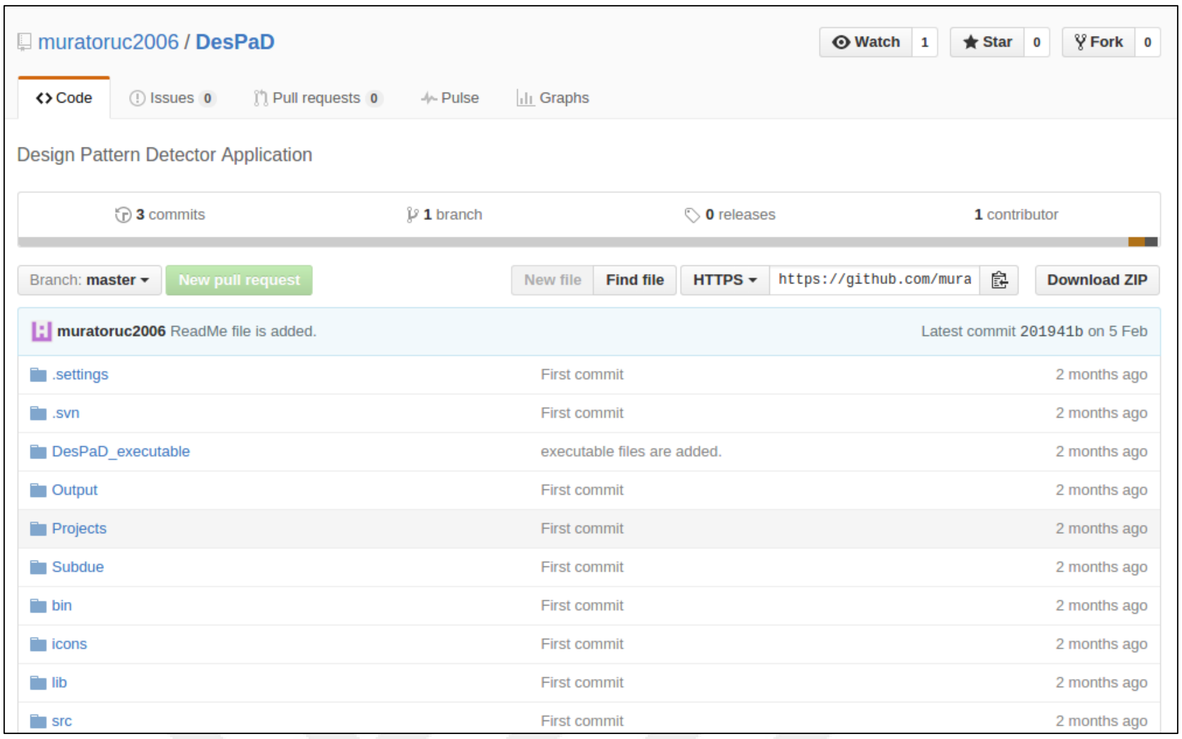
ANTLR (Another Tool for Language Recognition) and Graphviz libraries were also used while developing DesPaD. ANTLR [20] is an auxiliary plug-in for visiting abstract syntax tree of source code. Graphviz [22] is a tool for drawing graphs. These additional libraries are defined particularly in Section 2. Some shell scripts based on Linux operating system are also used in DesPaD. For example, Subdue-sgiso algorithm is used within Java by using Java's shell script execution feature.

DesPaD is designed and developed to automate detecting design patterns in a software project. It takes input source code's path and generates output graphs used as design patterns in the given software project. The user interface of DesPaD and steps of design pattern detection process is described in the following section.

4.2. Using DesPaD Interface to Detect Design Patterns

The DesPaD tool is made for finding GoF design patterns by a sub-graph mining search method from a given source code written in Java. DesPaD runs on Linux-based operating systems since it uses Subdue, sub-graph mining tool during the detection process, which is available for Linux systems. However, any isomorphic library can be plugged into DesPaD with a little effort. Subdue's isomorphic algorithm was chosen due to its convenience based on performance.

DesPaD can be downloaded from Github [29] for free as seen in Figure 4.1.


muratoruc2006 / DesPaD

Watch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Pulse | Graphs

Design Pattern Detector Application

3 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | New file | Find file | HTTPS | https://github.com/mura | Download ZIP

muratoruc2006 ReadMe file is added. Latest commit 201941b on 5 Feb

.settings	First commit	2 months ago
.svn	First commit	2 months ago
DesPaD_executable	executable files are added.	2 months ago
Output	First commit	2 months ago
Projects	First commit	2 months ago
Subdue	First commit	2 months ago
bin	First commit	2 months ago
icons	First commit	2 months ago
lib	First commit	2 months ago
src	First commit	2 months ago

Figure 4.1. DesPaD’s downloadable source code in Github

Once the DesPaD tool is downloaded and extracted to the local disk, Graphviz application must be installed because DesPaD uses it in its source code. To finish the installation of Graphviz the code below must be executed in terminal in Linux-based operating system.

- **sudo apt-get install graphviz**

Afterwards, by double-clicking the DesPaD’s shortcut, the program’s interface will show up.

Source Code Directory Path points to the root directory of a software project that is written in Java programming language. DesPaD will find all java files in chosen directory recursively and parse them in the background. For instance, when the /home/murat/software/source_codes/JUnit_3.8 folder is chosen as seen in Figure 4.3, all the source codes under that folder will be parsed automatically.

Program Directory Path refers the directory where the “DesignPatternDetection.jar” file exists. “DesignPatternDetection.jar” file is the executable file of DesPaD. DesPaD uses this directory for running shell script files in Subdue and creating output files after the sub-graph mining algorithm is completed. (See Figure 4.2)

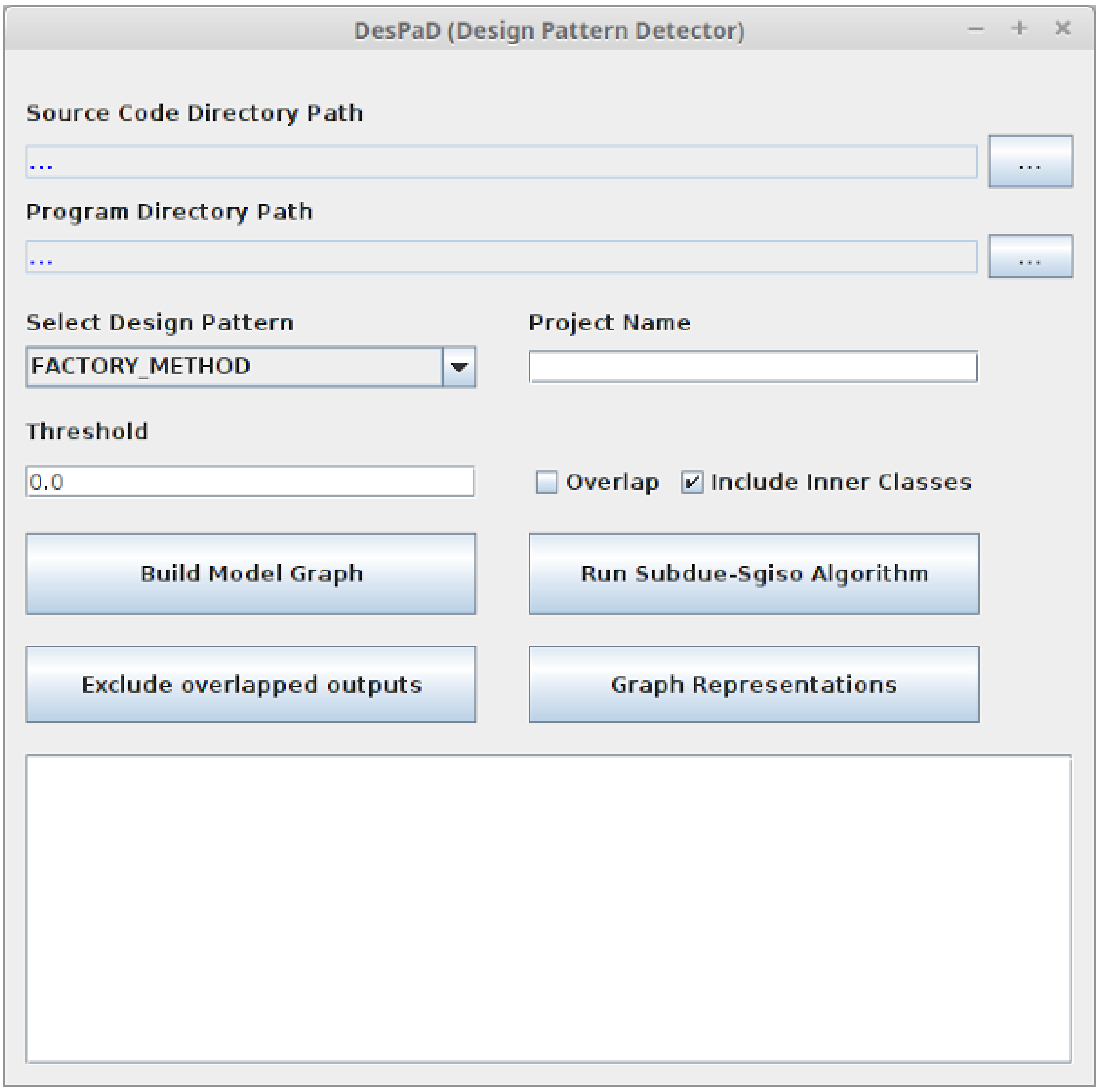
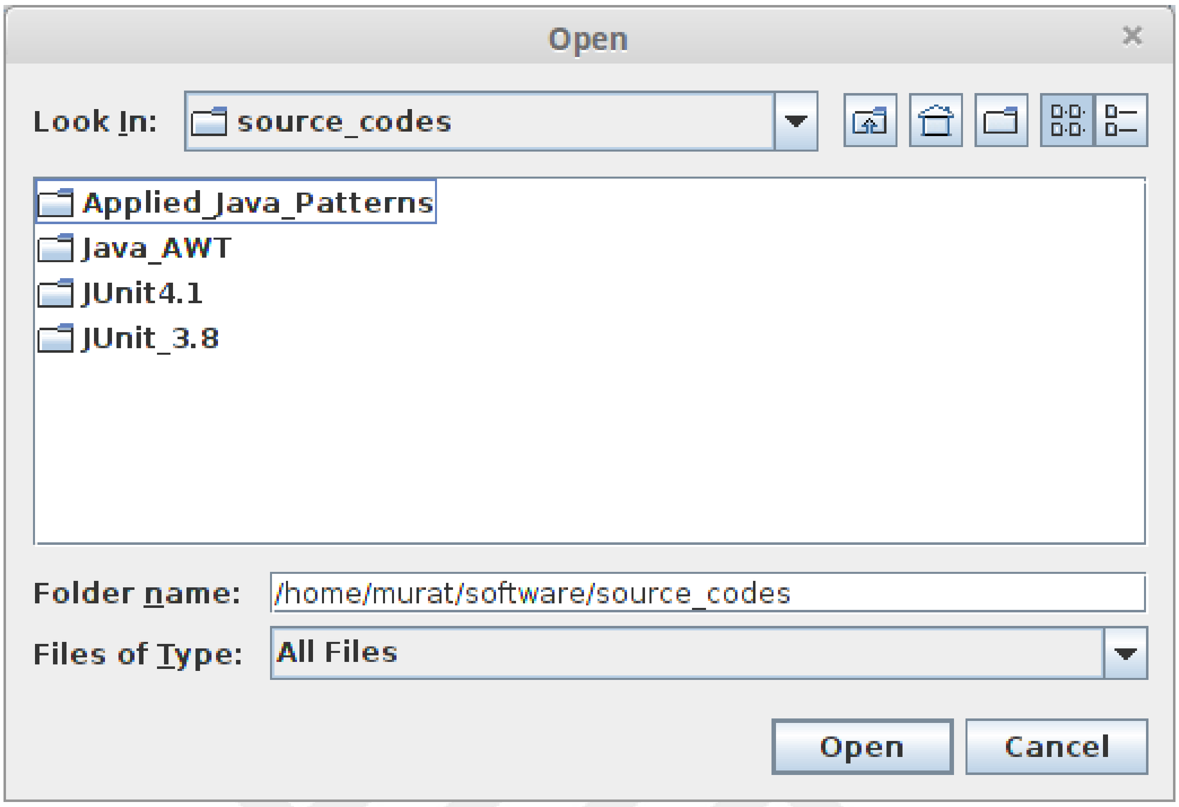


Figure 4.2. Interface of DesPaD

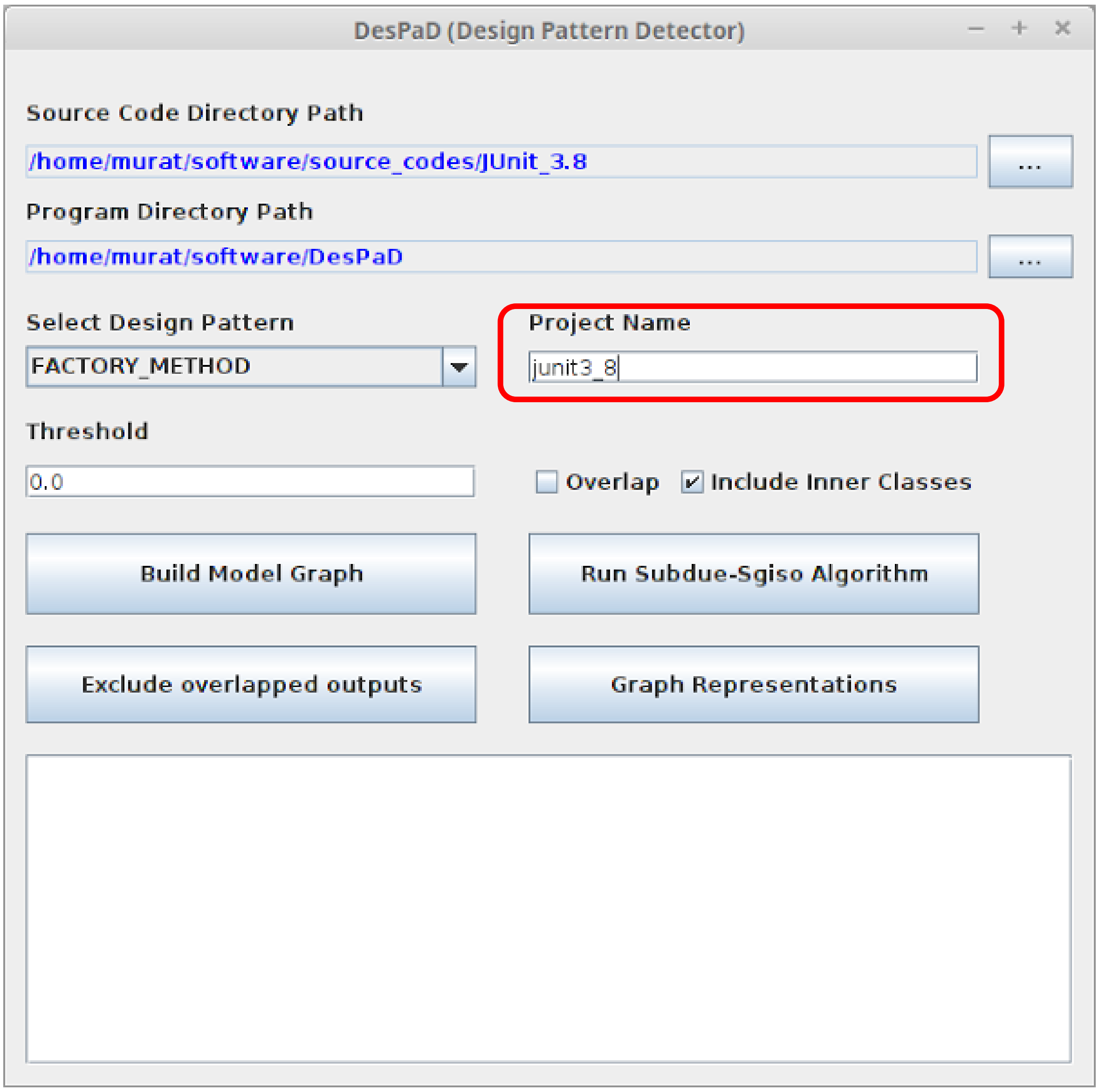
Select Design Pattern is a dropdown box, 23 GoF design pattern names are listed. One of them must be selected to be searched within source codes. Some of the patterns have more than one templates, this is because they cannot be defined by only one template. Those are decorator and template method patterns whose templates are seen in Figure 3.30, Figure 3.31 and Figure 3.70, Figure 3.71, respectively. Therefore, for detecting these patterns the program must be run more than once.



Folder name: /home/murat/software/source_codes

Figure 4.3. Selection of a directory interface

Project Name points the directory name in which output files of the program will be stored. It is convenient to give a name similar to the software project since it will be easier to find it later as seen in Figure 4.4.


DesPaD (Design Pattern Detector)

Source Code Directory Path

/home/murat/software/source_codes/JUnit_3.8

Program Directory Path

/home/murat/software/DesPaD

Select Design Pattern

FACTORY_METHOD

Project Name

junit3_8

Threshold

0.0

☐ Overlap ☑ Include Inner Classes

Figure 4.4. Project Name section

Threshold determines the similarity of found instances. Default value is 0.0. It means the detection algorithm will run as exact match. If for threshold similar patterns are desired, this value may between as 0.1 and 1.0. As the threshold value increases, the similarity of the found instances decreases.

Overlap has the default value of false. When it is checked, the detection algorithm will search as overlapped instances. This property is the internal parameter of Subdue.

Include Inner Classes is an option that determines whether DesPaD uses Java's internal classes or not while building the graph model. Its default value is true.

Once all properties are set accordingly, pattern detection can start.

The parsing of the given source code and forming its model graph process is executed by pressing “Build Model Graph” button.

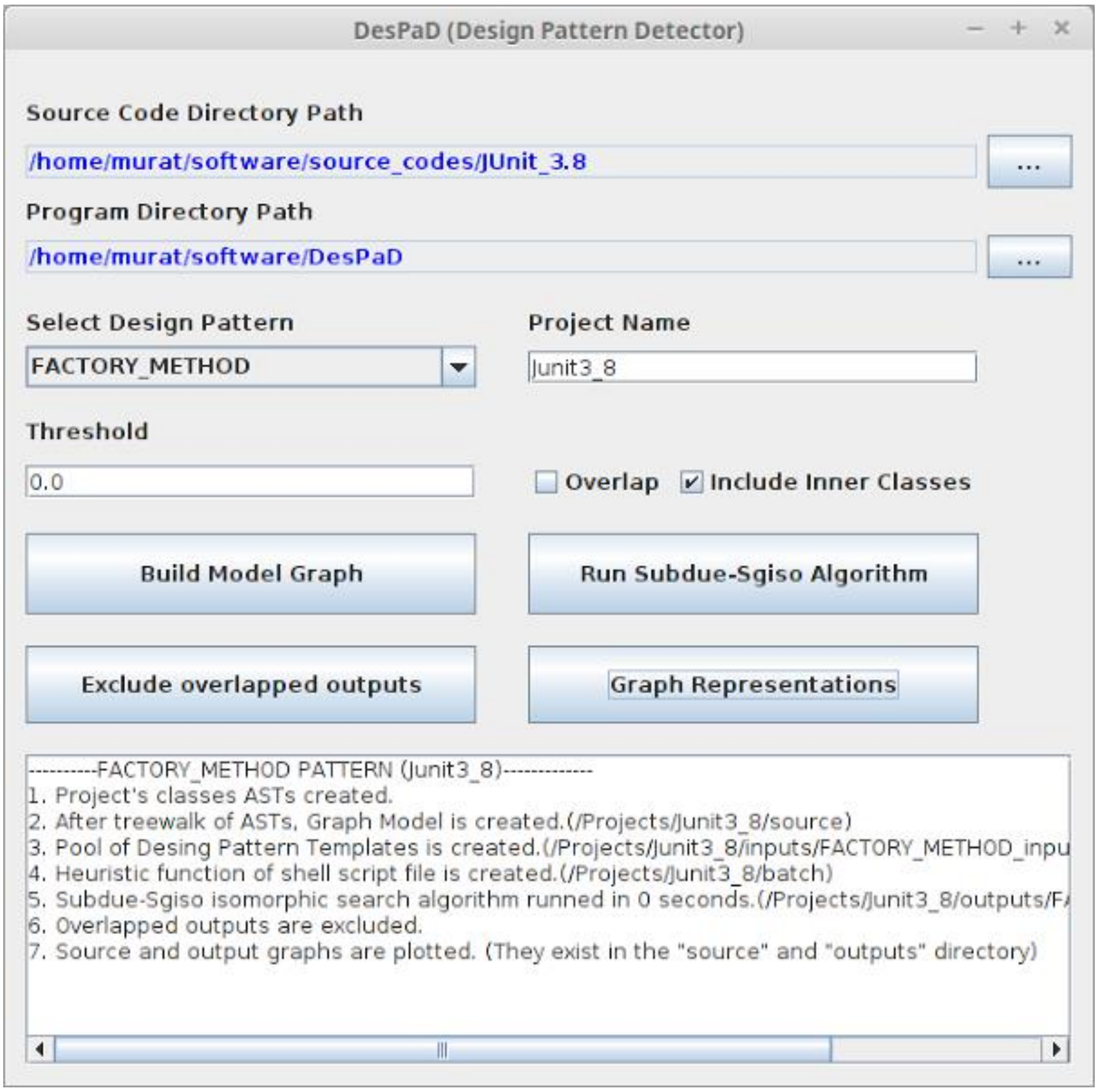
Pressing the “Run Subdue-Sgiso Algorithm” button involves the sub-graph mining algorithm. Isomorphic sub-graph mining search is applied in sgiso method in Subdue. The output files are saved automatically in the “Projects” folder in the selected “Program Directory Path” directory.

When the “Exclude overlapped outputs” button is pressed, the tool finds the overlapped instances in the outputs and excludes them automatically.

The tool plots the model graph of the given source code and the detected design patterns’ outputs as graphs by pressing the “Graph Representations” button. It uses the Graphviz open-source application for drawing.

During executing every step of the detection process, a proper log message is generated and displayed in the bottom pane of the tool to indicate the state of the process as seen in Figure 4.5.

Figure 4.6 shows four files created automatically by DesPaD in the selected “Program Directory Path” directory. Input, output and source files include text and graph representations of input graphs, detected graphs and model graph. Moreover, batch file exists shell script file to be executed for Subdue. This shell script is prepared according to the function defined in Section 3.3, Algorithm 1.


DesPaD (Design Pattern Detector)
Source Code Directory Path
/home/murat/software/source_codes/JUnit_3.8
...
Program Directory Path
/home/murat/software/DesPaD
...
Select Design Pattern
FACTORY_METHOD
Project Name
Junit3_8
Threshold
0.0
Overlap
Include Inner Classes
Build Model Graph
Run Subdue-Sgiso Algorithm
Exclude overlapped outputs
Graph Representations

```
----------FACTORY_METHOD PATTERN (Junit3_8)-------------
1. Project's classes ASTs created.
2. After treewalk of ASTs, Graph Model is created.(/Projects/Junit3_8/source)
3. Pool of Desing Pattern Templates is created.(/Projects/Junit3_8/inputs/FACTORY_METHOD_inpu
4. Heuristic function of shell script file is created.(/Projects/Junit3_8/batch)
5. Subdue-Sgiso isomorphic search algorithm runned in 0 seconds.(/Projects/Junit3_8/outputs/F
6. Overlapped outputs are excluded.
7. Source and output graphs are plotted. (They exist in the "source" and "outputs" directory)
```

Figure 4.5. The user interface after program is run

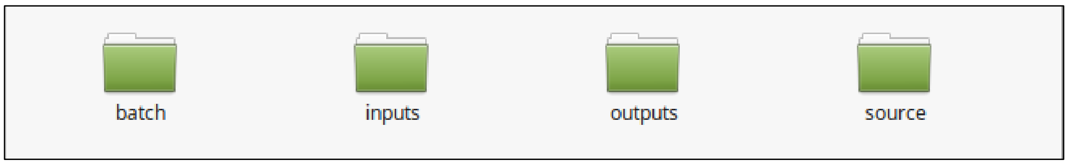


Figure 4.6. The directories created by DesPaD

4.2.1. Bridge Design Pattern Example

This section gives an example regarding how to create a template for a design pattern. The bridge design pattern is chosen as the example. Intent of the bridge pattern is described in Section 3.4.2. A template for the bridge pattern is created by analyzing bridge pattern’s class and sequence diagrams.

There are three steps for building the template.

- First, the nodes are determined by inspecting the class diagram seen in Figure 4.7. That is, entities listed in Table 3.1 are extracted. These are Abstraction, Implementor, Refined Abstraction and ConcreteImplementorA and ConcreteImplementorB together. These entities are mapped into Class 3, Interface 2, Class 4 and Class 1, respectively.
- Second, the relations are formed by analyzing both class and sequence diagrams given in Figure 4.7 and Figure 4.8. In this example, while ‘implements’ and ‘extends’ are captured from the class diagram, ‘related with its method’ relation is extracted from the sequence diagram of the bridge design pattern.
- Third, the nodes are tagged with 1,M and N. M and N means that the node and its specific relation may occur multiple times. In the example, the nodes of Class 1 and its ‘implements’ relation, Class 4 and its ‘extends’ relation are tagged with M and N from the class diagram of the related pattern.

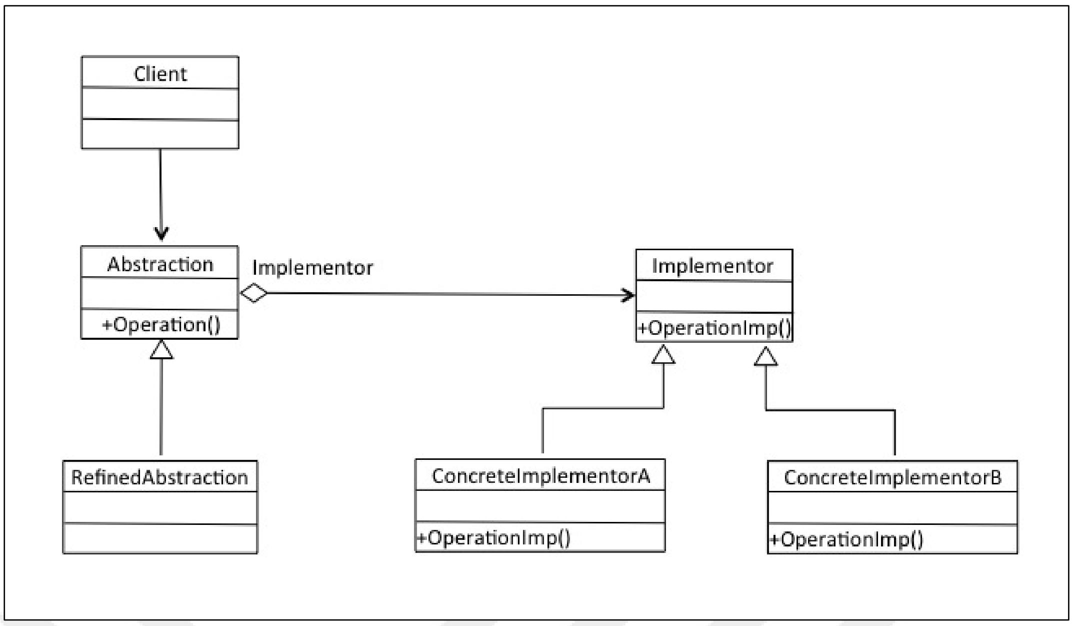


Figure 4.7. Class diagram of the the Bridge Design Pattern

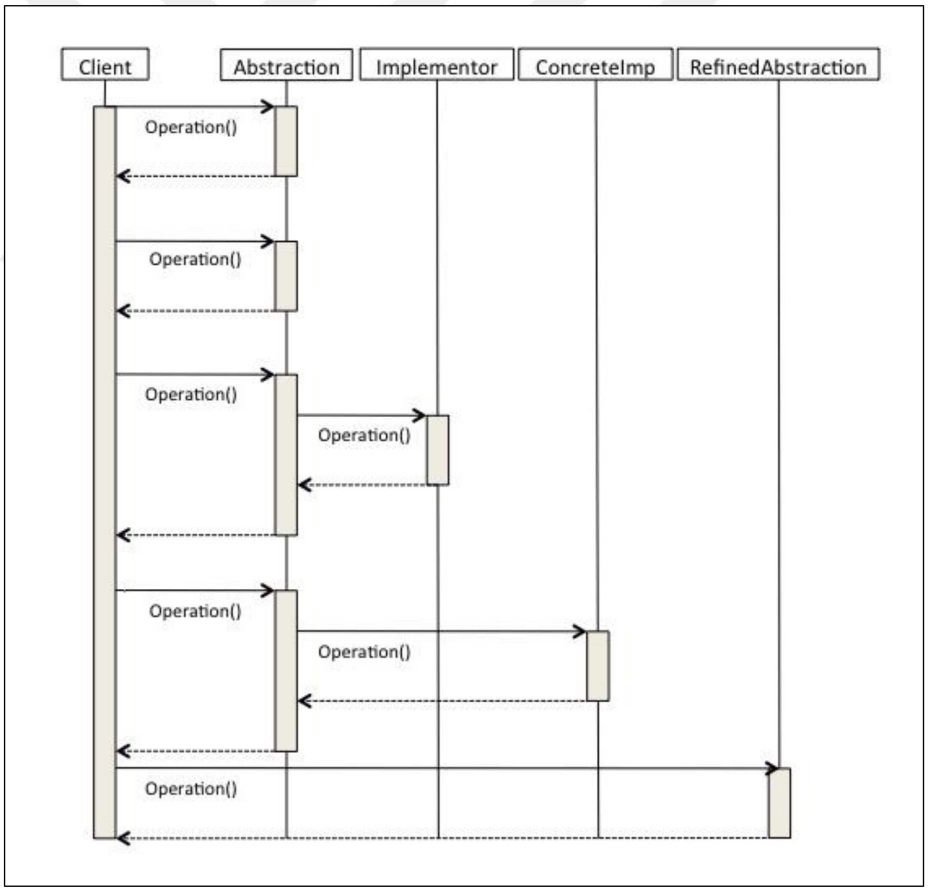


Figure 4.8. Sequence diagram of the Bridge Design Pattern

The resulting template is seen in Figure 4.9.

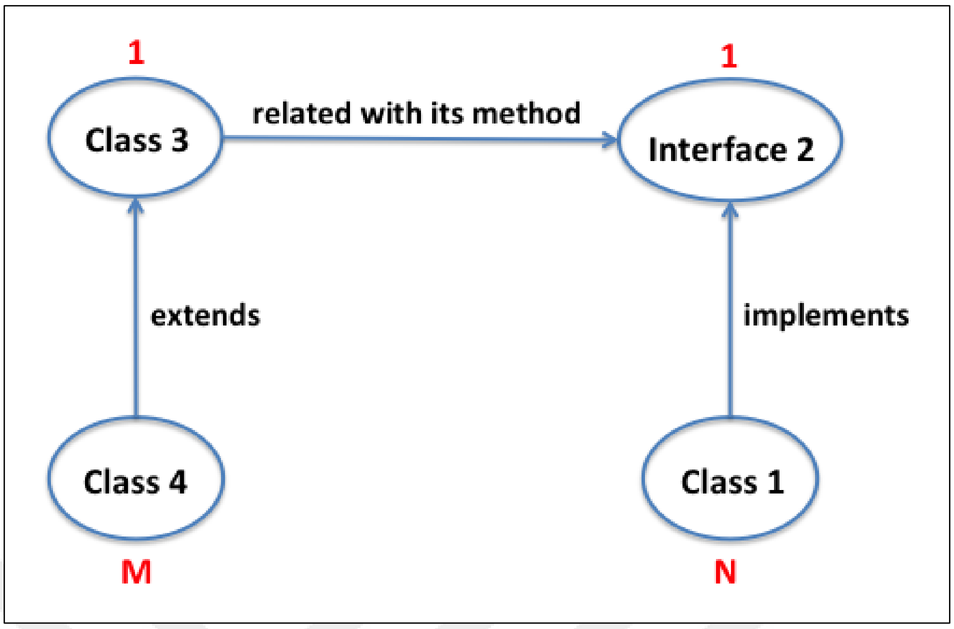


Figure 4.9. Template of Bridge Design Pattern

Figure 4.10 and Figure 4.11 give the example input files for bridge pattern which is in an appropriate format for Subdue, graph mining tool. Figure 4.10 shows an example where M and N are both equal to 1. According to the template in Figure 4.9, class 3 is extended and interface 2 is implemented only once. Similarly, Figure 4.11 shows an example where M equals to 3 and N equals to 2. According to the template in Figure 4.9, class 3 is extended three times and interface 2 is implemented twice. DesPaD generates these files dynamically and gives them for executing search algorithm to Subdue tool for detecting any Bridge design pattern in the given software project.

```
v 1 Class
v 2 Interface
v 3 Class
v 4 Class
e 1 2 implements
e 4 3 extends
e 3 2 related_with_its_method
```

Figure 4.10. An example base input file for Bridge Pattern(M=1 and N=1)

```
v 1 Class
v 2 Interface
v 3 Class
v 4 Class
v 5 Class
v 6 Class
v 7 Class
e 1 2 implements
e 5 2 implements
e 4 3 extends
e 6 3 extends
e 7 3 extends
e 3 2 related_with_its_method
```

Figure 4.11. An example input file for Bridge Pattern(M=3 and N=2)

5. EVALUATIONS

Extensive experiments are conducted to validate the approach proposed in this thesis. Results obtained through the DesPaD tool are compared against the closest rivals, PINOT [5], HEDGEHOG [13], FUJABA [14] and DP-Miner tool [15], which were mentioned in Section 2.

5.1. Experimental Setup

As test bed, source codes that were used as benchmarks by the rivals are used in the experiment to be compatible. These are demo source codes from “Applied Java Patterns (AJP) text book” [23] and source codes of three open-source projects, i.e. JUnit 3.8 [26], JUnit 4.1 [26] and Java AWT 1.3 [27, 28]. Projects in the test bed are all Java projects. Numbers of classes and lines of code regarding experimental projects are given in Table 5.1.

Note that DesPaD’s approach is not bound to a specific project. DesPaD can be adapted for another object-oriented programming language, e.g. C++ or C#. For doing this, the only part that should be changed or adapted is the one that creates the AST of that specific language.

Experiments were performed on a Linux running quad-core CPU commodity computer with 8 GB of RAM.

Table 5.1. Size of selected projects

Project	Number of Classes	Thousands of lines of code (KLOC)
JUnit 3.8	54	4.7
JUnit 4.1	157	4
AWT 1.3	407	102

5.2. Results

Evaluation results are analyzed in terms of precision and recall. Precision is the rate of true pattern instances found out of the total number of instances extracted by the tool. Recall is the rate of the true pattern instances found by the tool in the actual existing pattern instances. Actual true instances are based on the documentation of the open-source projects [26, 27, 28].

First, DesPaD is compared against similar tools in terms of capabilities. Table 5.2 shows which patterns in the AJP (Applied Java Patterns) example can be detected by each tool. The AJP example is chosen since it contains all GoF design patterns. Patterns are grouped as *creational, structural* and *behavioral* in the table. *OK* means that pattern can be detected by the tool. *X* means that the tool has failed to detect that pattern. If the tool does not cover the pattern at all, it is showed with a dash “-“ symbol. According to Table 5.1, DesPaD is the only tool, which can detect all 23 GoF patterns (100 %). The closest rival, PINOT can only detect 17 out of 23 patterns (74%).

Table 5.2. Comparison of verification of design patterns

	Tools			
	PINOT	HEDGEHOG	FUJABA	DesPaD
Creational				
Abstract Factory	*OK*	*OK*	*X*	*OK*
Builder	-	-	-	*OK*
Factory Method	*OK*	*OK*	*X*	*OK*
Prototype	-	*X*	-	*OK*
Singleton	*OK*	*OK*	*OK*	*OK*
Structural				
Adapter	*OK*	*OK*	*X*	*OK*
Bridge	*OK*	*OK*	*OK*	*OK*
Composite	*OK*	*OK*	*X*	*OK*
Decorator	*OK*	*OK*	*X*	*OK*
Facade	*OK*	-	*OK*	*OK*
Flyweight	*OK*	*OK*	*X*	*OK*
Proxy	*OK*	*OK*	-	*OK*
Behavioral				
CoR	*OK*	-	*X*	*OK*
Command	-	-	-	*OK*
Interpreter	-	-	-	*OK*
Iterator	-	*OK*	*X*	*OK*
Mediator	*OK*	-	*X*	*OK*
Memento	-	-	*X*	*OK*
Observer	*OK*	*OK*	*X*	*OK*
State	*OK*	*X*	-	*OK*
Strategy	*OK*	*OK*	*OK*	*OK*
Template Method	*OK*	*OK*	*OK*	*OK*
Visitor	*OK*	*OK*	-	*OK*

Second, DesPaD is tested against the open-source projects that are in the test bed.

Test results of DesPaD against JUnit 3.8, JUnit 4.1 and Java AWT 1.3 projects are seen in Table 5.3, Table 5.4 and Table 5.5, respectively. *Actual Instances* are the number of times a pattern really occurs in the source code. *Found instances* is the number of patterns that was returned by DesPaD and claimed as found in the source code of the software. True instances are the number of correctly found patterns by DesPaD.

According to test results, design patterns are detected with 80% precision and 88% recall values in average. DesPaD works almost perfect for the smallest project in the test bed, i.e. JUnit 3.8. As the number of classes and lines of codes in projects increase, precision and recall values may suffer. However, 78% of the actual patterns are still correctly detected and, for precision values below average, only the 21% of the cases generates false positives.

Table 5.3. JUnit 3.8 test results (DesPaD tool)

Pattern Name	Found/True Instances	Actual Instances	Precision	Recall
Bridge	2/2	2	100 %	100 %
Composite	1/1	1	100 %	100 %
Decorator	1/1	1	100 %	100 %
Singleton	0/0	0	NA	NA
Template Method	12/11	11	92 %	100 %

Table 5.4. JUnit 4.1 test results (DesPaD tool)

Pattern Name	Found/True Instances	Actual Instances	Precision	Recall
Bridge	4/1	1	25 %	100 %
Composite	2/2	2	100 %	100 %
Decorator	1/1	4	100 %	25 %
Singleton	4/1	1	25 %	100 %
Template Method	22/20	20	91 %	100 %

Table 5.5. Java AWT test results (DesPaD tool)

Pattern Name	Found/True Instances	Actual Instances	Precision	Recall
Bridge	20/20	30	100 %	66 %
Composite	9/2	2	22 %	100 %
Decorator	7/7	7	100 %	100 %
Singleton	18/14	14	78 %	100 %
Template Method	55/55	128	100 %	43 %

Third, DesPaD is compared with PINOT as it is identified as the closest work in literature in terms of capabilities. In addition, HEDGEHOG [13] or FUJABA [14] are not accessible in terms of test results on the chosen source codes. They are not available to play with, either. Therefore, it was not possible to compare with DesPaD. DP-Miner [15] provides promising results similar to DesPaD. However, since it does not cover all design patterns and uses a hard-coded mechanism by using specific properties of design patterns and related programming language, it was not considered for comparison with DesPaD.

Table 5.6 compares precision and recall performances of DesPaD and PINOT tools when they tested against the Java AWT 1.3 source codes. Regarding precision, PINOT performs 8% better than DesPaD in average. However, recall values are 47% better for DesPaD in average. That is, DesPaD detects much more design patterns than PINOT does. Recall values for PINOT can be as low as 3% while it can be only 35% in average where it is 82% for DesPaD.

Table 5.6. DesPaD vs. PINOT

Pattern Name	Precision		Recall	
	DesPaD	*PINOT*	*DesPaD*	*PINOT*
Bridge	100 %	75 %	66 %	10 %
Composite	22 %	67 %	100 %	100 %
Decorator	100 %	100 %	100 %	43 %
Singleton	78 %	100 %	100 %	22 %
Template Method	100 %	100 %	43 %	3 %
Averages	80	88	82	35

As explained before, it is not possible to compare DesPaD with DP-Miner since DP-Miner does not cover all patterns as DesPaD did. Table 5.7 gives precision and recall values for DP-Miner similar to what is presented in Table 5.4 for DesPaD in the JUnit 4.1 source codes. When looked at average precision and recall values, DesPaD performs 2% and 12% better than DP-Miner.

Table 5.7 DesPaD vs. DP-Miner

Pattern Name	**Precision**		**Recall**	
	DesPaD	*DP-Miner*	*DesPaD*	*DP-Miner*
Bridge	25 %	33 %	100 %	100 %
Composite	100 %	100 %	100 %	100 %
Decorator	100 %	100 %	25 %	100 %
Singleton	25 %	0 %	100 %	0 %
Template Method	91 %	100 %	100 %	65 %
Averages	69	67	85	73

Finally, the performance evaluation of DesPaD in terms of run time is done. Run times required to detect five different design patterns within the chosen open-source projects are seen in Table 5.8. In the table, input file count and time columns specify the number of files searched for in the model graph and total time for detection process, respectively. For instance, it takes 0,008 seconds to detect the bridge pattern in JUnit 3.8 project. As isomorphic sub-graph mining search is an NP-Complete problem, it is a big challenge to reach a good performance in case of large sized software projects. JUnit 3.8 and JUnit 4.1 are small or medium sized projects and DesPaD performs at the level of few seconds except for detecting the *Template Method* pattern. Java AWT 1.3, on the other hand, is a relatively large project, where the performance of DesPaD varies from few minutes to few hours. Note that these evaluations are performed on a simple commodity computer with limited CPU and memory. In case of a more powered experimental infrastructure, numbers for AWT 1.3 evaluations can be pulled down. Optimizing DesPaD's algorithm to get better results is left for future work.

Table 5.8. Performance test results in seconds

	JUnit 3.8		JUnit 4.1		AWT 1.3	
	Time	*Input file count*	*Time*	*Input file count*	*Time*	*Input file count*
Bridge	0,008	9	1	66	10560	690
Composite	0,07	9	3	36	9872	900
Decorator	0,05	18	1	132	950	1380
Singleton	22	1	2	1	5	1
Template	2	10	4758	42	4690	90

6. CONCLUSIONS

6.1. Research Contribution

This thesis provides a framework to search design patterns in a given source code by using an isomorphic subgraph mining algorithm.

Within this work, a high-level model graph out of source codes of a given project is built, representative graphs for design patterns are generated and those patterns were searched (and found if exist) in the model graph by using a subgraph mining algorithm.

To automate all this work, a detection tool called DesPaD is developed. It is tested against source codes from four different projects and compared it with the related work. To the best of latest knowledge, DesPaD is currently the only tool, which can detect all GoF design patterns. Also, it outperforms its closest work by creating 47% better recall values.

Note that, performance comparisons between DesPaD and PINOT or DP-Miner were not possible since they are not available to play with. Implementing those tools to make the comparisons is not in the scope of this thesis.

6.2. Future Work

As future work, optimizing the approach used in this thesis is intended. Due to the complexity of the sub-graph search algorithms, DesPaD's performance might suffer in case of large sized projects. To alleviate this problem, some partitioning or optimization algorithms will be investigated. Additionally, analyzing software metrics of the given source code and using them in the pattern detection process for optimization, is planned in the future.

The templates of patterns are formed as signatures of GoF design patterns up to the class and sequence diagrams. Despite all these, input graphs of the templates of design patterns may not be unique or may cover some other motifs in different domains. This drawback is one point that is planned to study in the future. In addition to using the class and sequence diagrams, forming signatures or fingerprints of the design patterns by using various features like software or graph metrics is one of the goals in future work.

Accordingly, detecting design patterns, which are described in some novel catalogues, will become another study of future work.

REFERENCES

[1] C. Gravino, M. Risi, G. Scanniello, G. Tortora, Does the documentation of design pattern instances impact on source code comprehension ? Results from two controlled experiments, Proceedings of the Working Conference on Reverse Engineering, IEEE CS, pp. 67-76, **2011.**

[2] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Trans. Software Engineering 28 (6), pp. 595-606, **2002.**

[3] C. Larman, Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall, **2001.**

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, **1995.**

[5] N. Shi, R. A. Olsson, Reverse Engineering of Design Patterns from Java Source Code, 21st IEEE ınternational Conference on Automated Software Engineering (ASE’06), **2006.**

[6] H. Lee, H. Youn, E. Lee, A Design Pattern Detection Technique that Aids Reverse Engineering, International Journal of Security and its Applications Vol. 2, No. 1, **2008.**

[7] Y. G. Gueheneuc, H. Sahraoui, F. Zaidi, Fingerprinting Design Patterns, Proceedings of the 11th Working Conference on Reverse Engineering (WCRE’04), **2004.**

[8] G. Antoniol, G. Casazza, M. Di Penta, R. Fiutem, Object-oriented design patterns recovery, The Journal of Systems and Software 59, pp. 181-196, **2001.**

[9] Y. G. Gueheneuc, P-MARt : Pattern-like Micro Architecture Repository, Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories (EPFPR), **2007.**

[10] Y. G. Gueheneuc, G. Antoniol, DeMIMA : A Multilayered Approach for Design Pattern Identification, IEEE Transactions on Software Engineering, Vol. 34, No. 5, **2008.**

[11] U. Tekin, F. Buzluca, A graph mining approach for detecting identical design structures in object-oriented design models, Science of Computer Programming, **2013.**

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32, pp. 176-192, **2006.**

[13] Dong, J., Sun, Y., Zhao, Y., Design Pattern Detection by Template Matching, Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Brazil, pp. 765-769, **2008.**

[14] M. A. Soliman, I. A. M. ElMeddah and A. M. Wahba, Patterns Mining from Java Source Code, Int.J. of Software Engineering, IJSE Vol.4 No.2, **2011.**

[15] R. S. Rao, M. Gupta, Design Pattern Detection by a Heuristic Graph Comparison Algorithm, International Journal of Advanced Research in Computer Science and Software Engineering 3(11), pp.251-255, **2013.**

[16] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design Pattern Detection, Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), **2003.**

[17] Subdue, http://ailab.wsu.edu/subdue/.

[18] U. Tekin, U. Erdemir, F. Buzluca, Mining Object-Oriented Design Models for Detecting Identical Design Structures, Sixth International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, pp. 43-49, **2012.**

[19] K. Ruohonen, Graph Theory Lecture Notes, **2013.**

[20] T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, **2012.**

[21] BNF Index of Java language grammar, http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html.

[22] GraphViz, www.graphviz.org.

[23] S. Stelting and O. Maassen, Applied Java Patterns, Prentice Hall, Palo Alto, California, **2002.**

[24] JUnit, http://www.junit.org/.

[25] Java AWT, http://docs.oracle.com/javase/7/docs/api/java/awt/.

[26] E. Gamma, JUnit A Cook's Tour, http://junit.sourceforge.net/doc/cookstour/cookstour.htm.

[27] C. Sars, P. Wessman, and M. Halme Design Patterns and the Java AWT, http://www.niksula.hut.fi/~ged/DesignPatterns/.

[28] J. Zukowski, Java AWT Reference, http://oreilly.com/catalog/javawt/book/index.html.

[29] DesPaD Design Pattern Detection Tool
https://github.com/muratoruc2006/DesPaD.git.

[30] Grady Booch, Draw me a Picture, In IEEE Software 28(1): 6-7, **2011.**

[31] U.R. Aktas, An Improved Graph Mining Tool and Its Application to Object Detection in Remote Sensing, Middle East Technical University, **2013.**

[32] Olmos, Ivan, Jesus A. Gonzalez, and Mauricio Osorio, Subgraph Isomorphism Detection Using a Code Based Representation, *FLAIRS Conference*. **2005.**

[33] C. Gravino, M. Risi, G. Scanniello, G. Tortora, Does the documentation of design pattern instances impact on source code comprehension ? Results from two controlled experiments, Proceedings of the Working Conference on Reverse Engineering, IEEE CS pp. 67-76, **2011.**

[34] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Trans. Software Engineering 28 (6) pp. 595-606, **2002.**

[35] Ellson, John, et al. "Graphviz—open source graph drawing tools." *Graph Drawing*. Springer Berlin Heidelberg, **2001.**

[36] JForm Designer 5.2, http://www.formdev.com.

[37] Applied Java Patterns,
http://authors.phptr.com/appliedjavapatterns/downloads.html.

[38] Grady Booch, Object-Oriented Analysis and Design With Applications, Benjamin Cummings, **1994.**

[39] Biuk-Aghai, Robert P., Object-Oriented Principles.

[40] James Rumbaugh, Michael Blaha, William Pramerlani, Frederick Eddy, and William Lorenson. Object-Oriented Modeling and Design. Prentice Hall, Eaglewood Cliffs, NJ, **1991.**

[41] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, **1992.**

[42] Addison-Wesley, Reading, MA. NEXTSTEP General Reference: Release3, Volumes1 and 2, **1994.**

[43] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madeany. Designing and implementating Choices: An object-oriented system in C++. Communications of the ACM, 36(9):117-126, September **1993.**

[44] Daniel C. Halbert and Patrick D. O’Brien. Object-oriented development. IEEE Software, 4(5):71-79, September **1987.**

[45] William F.Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In Proceedings of the 21st Annual Computer Science Conference (ACM CSC ‘93), pages 66-73, Indianapolis, IN, February **1993.**

AUTHOR'S CV

Credentials :

Name, Surname	: Murat Oruç
Place of Birth	: Bursa, Turkey
Marital Status	: Married
E-mail	: muratoruc@hacettepe.edu.tr
Address	: Kazim Ozalp Mah.Kelebek Sok.No:27 \ 19 Cankaya, Ankara

Education :

- Bachelor of Science (B.Sc.), Systems Engineering, Turkish Military Academy, 3.42, 2002-2006.
- Certification Program, Information Systems, Turkish Military Academy Institute of Defense Sciences, 2010 – 2011.
- Master of Science (M.Sc.), Computer Engineering, Hacettepe University, 3.89, 2013 – Present.

Foreign Languages :

English : YDS, 2012, 93,75.

Work Experience :

- Turkish Armed Forces, Battalion Executive Officer, 2007 – 2010.
- Turkish Armed Forces, Software Developer, 2011 – 2014.
- Turkish Armed Forces, Database Administrator, 2014 – Present.

Areas of Experiences : -

Projects and Budgets : -

Publications:

Oruc, M., Akal, F., Sever, H., Detecting Design Patterns in Object-Oriented Models by Using a Graph Mining Approach, *CONISOFT 2016,4th International Conference on Software Engineering*, **2016.**

Oral and Poster Presentations : -

Qualifications: MCSE : Data Platform (2015-2018)

Programming Skills: SQL, Java, C#, Delphi, R.

HACETTEPE UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

THESIS/DISSERTATION ORIGINALITY REPORT

HACETTEPE UNIVERSITY

GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

TO THE DEPARTMENT OF COMPUTER ENGINEERING

Date: 12/04/2016

Thesis Title / Topic: A Graph Mining Approach For Detecting Design Patterns in Object-Oriented Design Models

According to the originality report obtained by myself/my thesis advisor by using the ***Turnitin*** plagiarism detection software and by applying the filtering options stated below on **12/04/2016** for the total of **93** pages including the a) Title Page, b) Introduction, c) Main Chapters, and d) Conclusion sections of my thesis entitled as above, the similarity index of my thesis is **10 %**.

Filtering options applied:

1. Bibliography/Works Cited excluded (EXCLUDED)
2. Quotes excluded (EXCLUDED)
3. Match size up to 5 words excluded (EXCLUDED)

I declare that I have carefully read Hacettepe University Graduate School of Science and Engineering Guidelines for Obtaining and Using Thesis Originality Reports; that according to the maximum similarity index values specified in the Guidelines, my thesis does not include any form of plagiarism; that in any future detection of possible infringement of the regulations I accept all legal responsibility; and that all the information I have provided is correct to the best of my knowledge.

I respectfully submit this for approval.

Date and Signature

12/04/2016

Name Surname : Murat Oruç

Student No : N13123316

Department : Computer Engineering

Program : Computer Engineering

Status : Master of Science

ADVISOR APPROVAL

APPROVED.

Asst. Prof. Dr. Fuat AKAL