Using Codecharts for formally modelling and automating detection of patterns with application to Security Patterns

By

Abdullah A. H. Alzahrani

A thesis submitted for the degree of PhD in Computer Science

School of Computer Science and Electronic Engineering

University of Essex

February 2016

Acknowledgements

My sincerest thanks go to Dr. Martin Colley for his invaluable support, generosity and benevolence. Through his supervision and guidance, I was able to concentrate on my PhD project, refine my work and critically evaluate my work and relevant works in the field.

I am also grateful to Ummul Qura University for sponsoring me and enabling me to pursue my postgraduate study at PhD level. In particular, I am deeply indebted to the Computer Science College at Al-Qunfudah.

My heartfelt gratitude and appreciation go to my mother (Moteebah) and my father (Abdurrahman) for their continuous support and help. I would also like to thank my brothers and sisters for motivating me to continue my study in a UK university.

I wholeheartedly thank my wife for always being by my side and for being there for me. Throughout my years of PhD study, she was a model of patience, perseverance and compassion. I should also apologise to the two jewels of my home, my two daughters Kinda and Sophia, for not being there to share with them some of their joyful moments.

Last but not least, I cannot forget my study-abroad companions, with whom I used to share my happiness and sorrow and to whom I often offloaded my concerns and worries.

Abstract

Software design patterns are solutions for recurring design problems. Many have introduced their catalogues in order to describe those patterns using templates which consist of informal statements as well as UML diagrams. Security patterns are design patterns for specific security problems domains, therefore, they are described in the same manner. However, the current catalogues describing security patterns contain a level of ambiguity and imprecision. These issues might result in incorrect implementations, which will be vital and at high cost security flaw, especially after delivery. In addition, software maintainability will be difficult thereafter, especially for systems with poor documentation. Therefore, it is important to overcome these issues by patterns formalisation in order to allow sharing the same understanding of the patterns to be implemented.

The current patterns formalisation approaches aim to translate UML diagrams using different formal methods. However, these diagrams are incomplete or suffer from levels of ambiguity and imprecision. Furthermore, the employed diagrams notations cannot depict the abstraction shown in the patterns descriptions. In addition, the current formalisation approaches cannot formalise some security properties shown the diagrams, such as system boundary.

Furthermore, detecting patterns in a source-code improves the overall software maintenance, especially when obsolete or lost system documentation is often the case of large and legacy systems. Current patterns detection approaches rely on translating the diagrams of the patterns. Consequently, the issue of detecting patterns with abstraction is not possible using such approaches. In addition, these approaches lack generality, abstraction detection, and efficiency.

This research suggests the use of Codecharts for security patterns formalisation as well as studying relationships among patterns. Besides, it investigates relationships among patterns. Furthermore, it proposes a pattern detection approach which outperforms the current pattern detection approaches in terms of generality, and abstraction detection. The approach competes in performance with the current efficient pattern detection approaches.

Table of Contents

List of Figures

List of Tables

List of Acronyms

SAP	Single Access Point pattern
CP	Check Point pattern
TTP Toolkit	Two-Tier Programming Toolkit
JAAS	Java Authentication and Authorization Service
LePUS3	LanguagE for Patterns Uniform Specification

List of Publications

Alzahrani, Abdullah AH, Amnon H. Eden, and Majd Zohri Yafi. "Conformance checking of Single Access Point pattern in JAAS using Codecharts." *Information Technology and Computer Applications Congress (WCITCA), 2015 World Congress on*. IEEE, 2015.

Alzahrani, A.A.H., A.H. Eden, and M.Z. Yafi. 2014. 'Structural Analysis of the Check Point Pattern'. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*, 404–8. doi:10.1109/SOSE.2014.56.

Alzahrani, Abdullah A. H., Majd Zohri Yafi, and Fawaz K. Alarfaj. 2014. 'Some Considerations on UML Class Diagram Formalization Approaches'. *International Journal of Computer, Information, Systems and Control Engineering* 8 (5): 47 – 50.

Alzahrani, Abdullah A. H., and Amnon Eden. 2013. 'UML versus LePUS3 for Modelling Structure of Security Patterns'. In, 260–69. Kyiv, UA: Cybernetics Faculty of Taras Shevchenko National University of Kyiv.

1 Introduction

1.1 Overview

In software engineering, design patterns are solutions for recurring design problems. They have been introduced and developed to convey expertise of specialists in a convenient and clear way. One type of these patterns is security patterns which have been first introduced by Yoder and Barcalow [1], [2] in 1997. However, security patterns have not been considered sufficiently and there is a lack of research in this direction, especially in terms of formalisation, reasoning, and detection of security patterns.

It is important for security patterns to be implemented correctly in order to be maintained and evolved. One way to achieve this is to detect them from the source code. Many researchers [3]–[8] have introduced their approaches to detect instances of design patterns from the source code. However, these pattern detection systems cannot detect security patterns from source codes.

The inability of pattern detection systems to detect security patterns from source codes might be attributed to the way patterns are represented in these systems. This throws light on a number of issues in the current systems, such as formalisation of patterns and generality in pattern detection approaches. These issues were a motivation for the research of this thesis.

Therefore, this research focuses on using Codecharts for modelling security patterns as Codecharts offer a visual and formal modelling method, unlike the current method (UML diagrams) of modelling security patterns. Furthermore, it aims to develop a new approach to detect patterns modelled in Codecharts from a source code. One of the goals of the patterns detection direction of this research is to develop a general patterns detection that, unlike others, allows for the detection of any design patterns as well as user-defined patterns. This is to overcome the issue of restricting the pattern detection to fixed pre-defined and hard-coded design patterns.

1.2 Motivation

While studying security patterns, many unsolved problems have been noticed. These problems start with the descriptions of the security patterns. An example is the ambiguity encountered in the descriptions. Ambiguity is one of the main reasons behind the lack of formal descriptions of the patterns. With the existence of this ambiguity, the solutions cannot be applied efficiently. So, even when solutions are implemented, the different understanding of solutions (caused by ambiguity in patterns descriptions) among people who are involved in the development, can lead to inefficiencies in maintaining, verifying, and documenting a software system. Furthermore, there is a chance of incorrect implementation of these solutions.

It is important to distinguish between the terms: informality, lack of precision, and ambiguity; which are used to describe the issues that exist in current security patterns catalogues. First, informality means that the current modelling notations used in the catalogues are not built on formal foundation. Specifically, UML diagrams are the modelling notations which are used in modelling these patterns in the catalogues. It is known that many have introduced their formal representations of these notations, however, there is not one agreed-on formal representation of these notations. In addition, the authors of the current catalogues do not provide a formal representation with the patterns descriptions.

Second, lack of precision means that the English statements and the modelling notations, which are used in the catalogues to describe the patterns, are unclear and needs further explanations. Many instances of this issue can be noticed in the catalogues, one example is the words "May", "Could", "Usually" etc., are always encountered in the security patterns descriptions. Another example is shown in Figure 1 which illustrates the modelling of Check Point pattern in [9]. When looking at the relation between *Client* and *ProtecedSystem*, it can be seen that the relation is an "*interacts with*", however, this does not imply any clear view on the type of interaction and who starts the interaction. Moreover, many other questions would be asked, for instance, is there any return values? Is there parameters allowed to be passed directly?

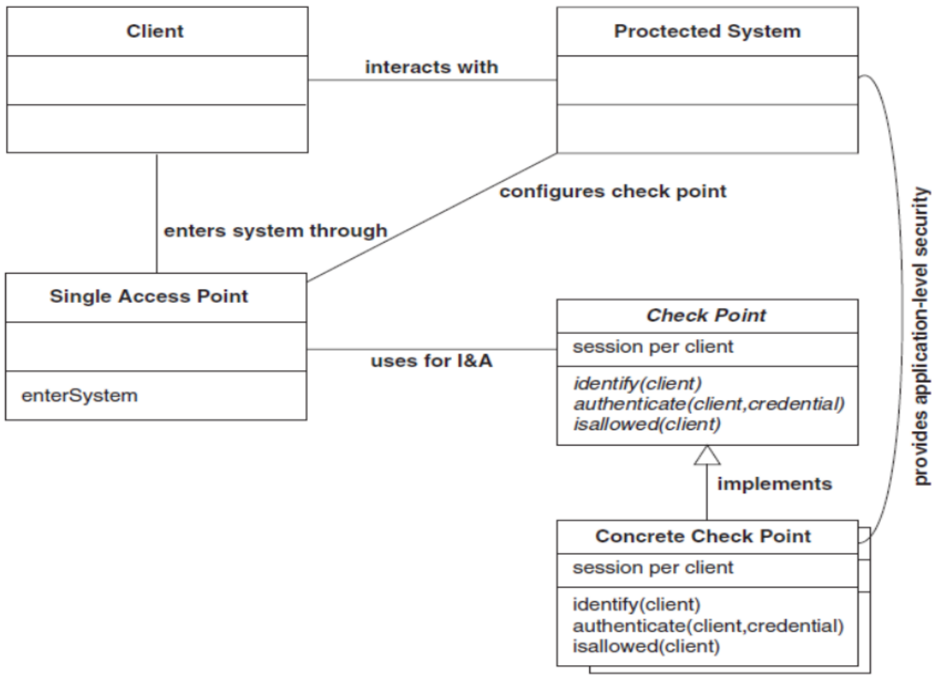


Figure 1: Lack of precision example: UML class diagram modelling Check Point pattern

Finally, ambiguity can be noticed in many of the existing security patterns descriptions. This issue and the above mentioned issues stand as an obstacle when detecting patterns or carrying out any rigorous reasoning on them. Ambiguous statements, in the pattern descriptions, misleads the pattern user as they carry more than one meaning. For example, Figure 2 illustrates the modelling of Protection Reverse Proxy pattern in [9] page 459. By looking at the diagram, two relations named "*getPage*" are modelled between *Firewall* and *ReverseProxy* classes. First, the relations flow can be from either class and can be understood differently by readers. Second, from the name of the relation, it can be realised that the relation is a call method, however, no operations but attributes are shown in the classes' modelling. So, it might be a relation of holding an instance of one another class.

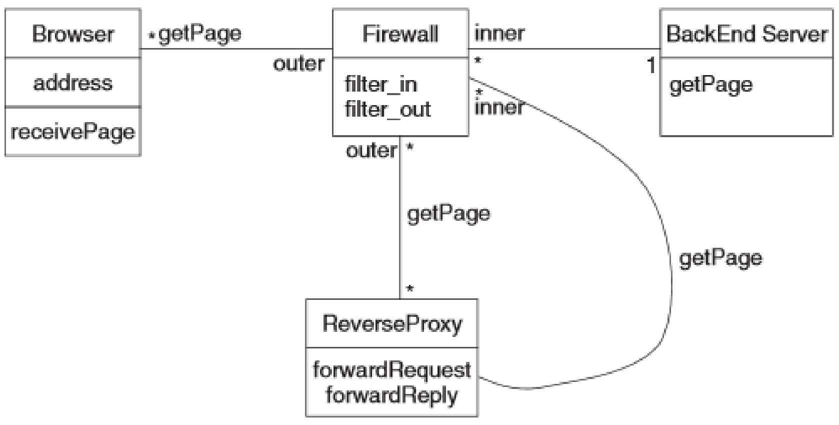


Figure 2: Ambiguity in relations example: UML class diagram modelling Protection Reverse Proxy pattern

Although security design patterns are a good approach to handling security design problems, false implementation of the security patterns may result in a security hole which might be used by an attacker [10]. With no formal and precise representation of security patterns, false implementation problems emerge. For example, Figure 3 illustrates the common case which can occur when no formal and precise representation for a security pattern is given. Therefore, formal representations of security design patterns are practically important.

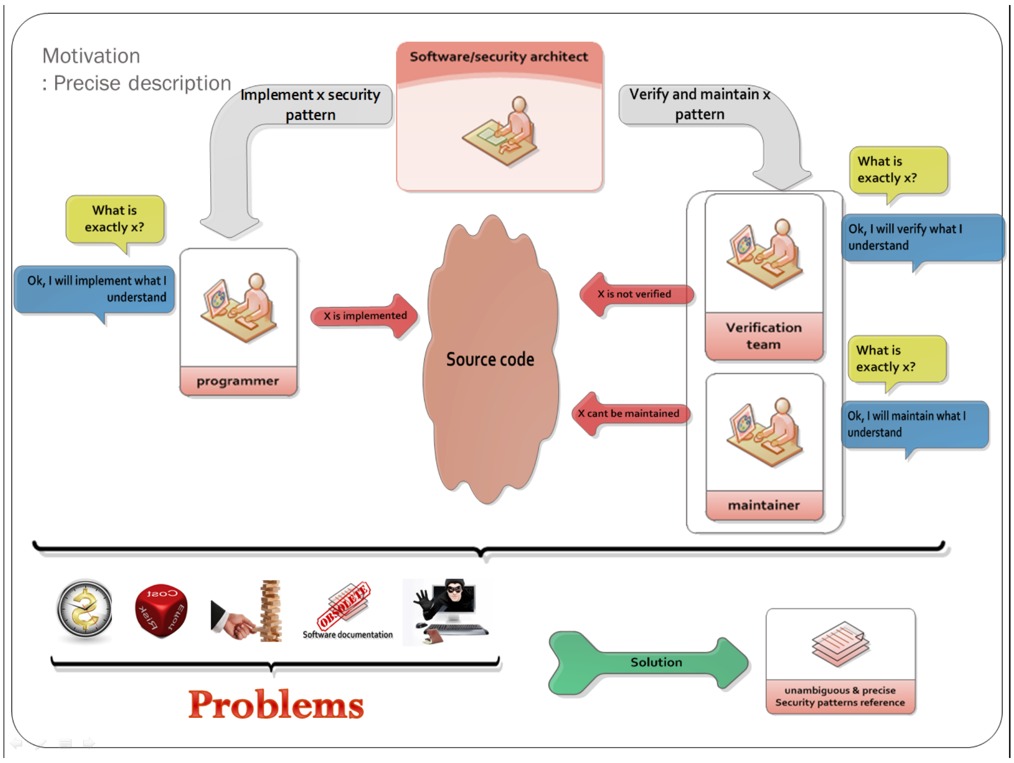

Motivation
: Precise description
Software/security architect
Implement x security pattern
Verify and maintain x pattern
What is exactly x?
Ok, I will implement what I understand
programmer
X is implemented
Source code
X is not verified
X cant be maintained
Verification team
maintainer
What is exactly x?
Ok, I will verify what I understand
What is exactly x?
Ok, I will maintain what I understand
Software documentation
Problems
Solution
unambiguous & precise Security patterns reference

Figure 3: Security patterns precise descriptions motivation

Inconsistency between design and implementation is also a common issue that many organisations encounter. The reason behind this issue lies in absent, obsolete, poor and/or lost software documentation. The issue of inconsistencies between design and implementation cannot be solved if detecting the patterns in the source code is not possible. Therefore, it is highly beneficial to have an approach which allows for detecting and locating the pattern within the source code by using high level analysis that saves both effort and time [10].

Figure 4 demonstrates the common situation when maintaining a system. For example, when a software architect has left an organisation which needs to maintain his or her software; they may think that he had implemented pattern x; however, the architect may not have left any proper documentation and this would lead to the need for dedicating people who can go through the source code in order to first understand it and then to locate the target pattern.

Another situation is the evolution of a legacy system which has been growing over the years and has never been documented properly. In this situation, developers, software architects and software designers need to know where the instances of x pattern are located. This is to ensure that the evolution is logical, systematic and effort saving.

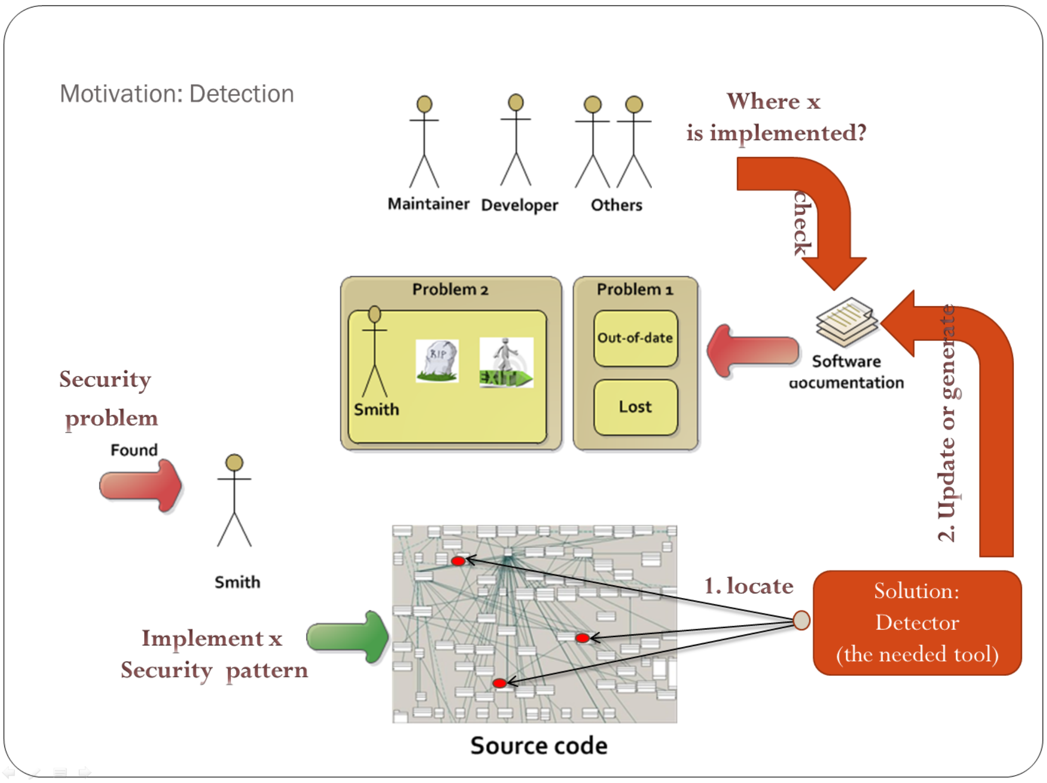


Figure 4: Security patterns detection motivation

So, if there is a formal representation for pattern x and a tool which automatically indicates the places of x in the source code of the software, this will allow the development people to efficiently maintain the software. In addition, it will save the time and efforts spent on the maintenance and evolution of the software. Moreover, detection design pattern instances will help maintain the documentation of the software as well as give the developers a chance to generate (in case software documentation has become obsolete) an up-to-date architectural documentation of the software as the major patterns can then be located.

Many reasons motivate searching and finding patterns instances in source codes of legacy software systems. One reason is to allow managers and different groups involved in the software development to check that the design has been implemented as agreed. Another reason is regarding the maintenance of the code as finding patterns in the source code helps in better understanding of the code and save efforts. In addition, in a case of obsolete or lost documentations, finding patterns in source code is a significant help in understanding unfamiliar code and re-generate software documentation. However, as this research focus on security patterns, detection is obstructed with the existence of the above mentioned issues (informality, lack of precision, and ambiguity) in security patterns catalogues. So, finding a way to overcome these issues and model the patterns formally, precisely, and unambiguously is highly important prior the detection of patterns.

In conclusion, the need for formal representation of security design patterns is highly important. By having such formal representation available, many will benefit, in particular software security architects, software designers, programmers and maintainers who will be confident that they share the same understanding of the patterns which are to be under consideration. Moreover, formal descriptions of security patterns will allow for developing algorithms to automate the process of detection.

1.3 Definitions of problems

- **Pattern formalisation**

The problem of security design patterns formalisation can be described, in general, as representing patterns in a formal manner that depicts the level of abstraction (which appears in patterns descriptions in their original catalogues) and security properties such as system boundary.

- **Patterns variants and relationships**

The problem of security design patterns variants can be described, in general, as studying and reasoning of the patterns which are claimed to be related or described in catalogues differently.

- **Pattern detection**

The problem of pattern (design or security design) detection can be described , in general, as looking for instances of static structure of security and/or design patterns, though with a level of abstraction which appears in patterns descriptions in their original catalogues.

From the aforementioned problems, it can be concluded that the problems are of choosing and using Codecharts to represent fully decidable patterns description statements in order to reason out the variants of a pattern described in catalogues differently and the relationships which are claimed to exist among patterns. In addition, it is the problem of looking for instances of Codecharts (specification in LePUS3) with variables in a source code.

This is challenging because there need to be theoretical and empirical investigations on pattern catalogues, particularly security patterns catalogues which suffer from ambiguity and imprecision. In addition, these investigations have not been carried out by any researchers. Furthermore, the Brute force solution of detection of patterns with a level of abstraction is exponential in complexity.

1.4 Goals / Objectives

Research goals

1. To use Codecharts and formalise security patterns, to investigate the relations among patterns, and to highlight the patterns variations according to different patterns descriptions;
2. To use the TTP Toolkit in order to check the conformance of intended implementations of patterns modelled in Codecharts in existing source codes;
3. To design and implement an efficient algorithm detecting instances of patterns specified in Codecharts.

Objectives

1. To compare the UML class diagrams for security and design patterns with corresponding ones in Codecharts and analyse differences conceptually;
2. To design an "efficient" pattern detection algorithm which detects static structure of patterns modelled in Codecharts from a source code within a reasonable amount of time;
3. To analyse the algorithm complexity and compare it with similar detection algorithms and brute force algorithm;
4. To extend the TTP toolkit to deliver an implementation of the detection algorithm that is practical in the following terms: "Find all of the instances of pattern *y* specified in Codecharts in program *z* within *s* seconds";

5. To investigate the accuracy and efficiency of the detection algorithm and identify the tool and possible improvements;
6. To carry out a number of case studies on open source software to evaluate composition of security and/or design patterns;
7. To study the relations between security and/or design patterns modelled in Codecharts;
8. To use Codecharts to highlight the variations in different patterns descriptions.

1.5 Research contributions

At present, many concerns have been raised in the field of security patterns. One instance is the ambiguity in the descriptions of security patterns. This issue has been addressed by many researchers. The researchers in [11] claim that due to the lack of formal descriptions of design motifs, their approach cannot be fully automated. In addition, The researchers in [12] state that having well-described and well-defined security patterns descriptions would improve application security when the programmers rely on these descriptions.

Another concern is the fully automatic detection of security and design patterns. This problem emerges in a common maintenance case when having obsolete, lost or/and poor documentation. Many have introduced their approaches to detect patterns from a source code. However, only a small number of these approaches count as successful. The major defect of these approaches is that they are not being fully automated due to many reasons such as the problem with informal specifications of the patterns to be detected.

Therefore, this research takes advantage of the formality and other properties of Codecharts to contribute to representing the structure of security patterns formally and building a formal reference for security patterns using Codecharts. As a result, building a reference would be the first attempt to formally model some of well-known security patterns.

In addition to the formal reference, this research contributes to developing an algorithm for detecting the patterns in the source code in a reasonable amount of time. This will solve the problem of automatically assuring the conformance of the design. As mentioned earlier in this report, the detection would help in the maintenance process as it can automate the process of finding the patterns in the source code in the case of obsolete, lost or/and poor documentation. Moreover, it would be an essential method for enhancing applications security when it is used to detect security patterns in particular, as it will allow for checking the consistency of the design to be implemented. Furthermore, this research aims to investigate the correctness, complexity and performance of the algorithm. A comparison of the algorithm with the existing ones in the field will be conducted. This will be considered as a key part of the research theoretical contribution.

Indeed, it is highly important to implement the pattern detection algorithm. The implementation of the algorithm will show the tangible benefits for the algorithm. Therefore, the research aims to produce a tool which implements the algorithm. The tool is intended to be integrated with the TTP toolkit [13]. This will represent the empirical contribution of the research.

The following list is to summarise the contributions of this research:

1. The research will theoretically investigate the use of Codecharts for formally modelling some of the well-known security patterns.
2. The research will employ this formal modelling for studying the security patterns variations when inconsistency occurs in descriptions

of patterns in different catalogues. In addition, it will employ the modelling for reasoning out a number of relationships which are claimed among patterns.

3. The research will develop an efficient pattern detection algorithm which aims to find the instances of given patterns that are modelled in Codecharts with a level of abstraction.
4. The research will conduct a set of empirical case studies on detection and conformance checking of a number of security and design patterns in open source codes.

1.6 Outline of the thesis

This thesis has been structured as follows. After introducing the research goals and motivations in Chapter 1, Chapter 2 focus on giving a broad background on the topics which relate to this research directions. Next, Chapter 3 demonstrates the conducted literature review on earlier studies on pattern formalization and on pattern detection. Next, Chapter 4 discusses the reasons of using Codecharts for security patterns modelling and describes the method used to model security patterns from their original catalogues. Chapter 5 motivates the idea of identifying security patterns variants using Codecharts and studying the relations among security patterns and with design patterns.

Chapter 6 shows and discusses a number of case studies on the manual finding and conformance checking of a number of security pattern. Chapter 7 explains the proposed patterns detection approach of this research. In addition, it shows a theoretical comparison of the proposed solution with the brute force solution of the problem of pattern detection using Codecharts. Next, Chapter 8 demonstrates a number of case studies on pattern detection approach and compares with the manual results shown in Chapter 6. Finally, Chapter 9 evaluates the pattern detection approach which has been proposed in this research with comparison with other pattern detection systems.

2 Background

2.1 Object-Orientation

Object orientation is a software design methodology which relies on representing a software system in the concept of object; which is an entity that encapsulates a set of data and related operations on them [14], [15]. A Class is regarded as a template for objects creation [16]. This design paradigm has been evolved radically especially after the introduction of UML (Unified Modelling Language), which is a now a widely used and accepted software design modelling language [17].

The object-oriented programming paradigm has been built on this design methodology and has been regarded as one of the most popular programing paradigms, such as logical, functional and imperative paradigms. A number of existing programing languages support object-oriented programing, for instance, Smalltalk, Java, C#, C++, and others. The object-oriented paradigm considers the generality and reusability of the design and code. In addition, many features can be gained when using the object-oriented programing paradigm. The main features are: inheritance, abstraction, dynamic binding and polymorphism [14].

Highlighting object-orientation is important in this research, which will be tackling the design patterns that are described using this paradigm. Furthermore, the design description language (Codecharts) chosen for patterns modelling in this research is an object-oriented description language. Moreover, it is important to state here that the GoF [18], which is the most popular design patterns catalogue, employs this paradigm in order to template the well-known design patterns. This catalogue has offered the current widely used design pattern template, which has also been used in security design patterns catalogues considered in this research.

In particular, this research aims to model some of the OO aspects shown in security patterns descriptions such as inheritance, abstract, and dynamic binding. For example, using Codecharts notations (Figure 5), *Hierarchy* and the *binary relation* (arrow between two class notations) labelled with *Inherit* are used to model inheritance aspect that might occur in descriptions. Another notation, which is used to model a class as an *Abstract*, is the *unary relation* (the tringle superimposed on a class). In addition, the aspect of dynamic binding is modelled with *Hierarchy* notation and *Signature* notation superimposed on it. The *Hierarchy* means that there is an abstract class which has a number of subclasses with inheritance relation with it, *Signature* of a method with same name in the abstract class and all subclasses. Later in this thesis, the modelling of these aspects will be shown in modelling of security patterns as well as the detection of these modelling.

2.2 Software maintenance

Software maintenance is regarded as the activities that are carried out on a software system after its release in order to fix problems, improve features, and/or help with adaption to the environment. These activities are usually done by a team that is different than the development team and aim to make changes in the system as required [19], [20].

The main two problems in software maintenance are high costs and considerable efforts. These issues are linked to the difficulties which maintainers face when trying to understand the software source code. Maintainers' difficulties in understanding will increase the effort, which will consequently increase the time and the costs spent [20]. In addition, understanding the software source code will improve the software comprehension. It is estimated that 50% of the maintenance effort is taken for software comprehension [21].

Codecharts is a visual, precise, and concise language, which has a few notations. When using such a language for documenting security design patterns or any design pattern, these features in the language will help ease the cognitive processes needed for software understanding [22], [23]. Therefore, this research will improve the overall software maintenance by improving the software maintainers' understanding.

Furthermore, searching and locating design patterns and arbitrary user-defined design patterns will improve software comprehension and software re-documentation [6], [23], which in turn will improve the overall software maintenance. This research aims to detect patterns instances within the source code. In addition, as using Codecharts, the research aims to take advantage of the visuality of Codecharts in order to show the existing instances of the searched-for patterns in the source code. As previously mentioned, this will improve the software comprehension [22], [23].

2.3 Design Patterns

A pattern is defined according to Longman dictionary as "*a shape used as a guide for making something*" [24]. In software engineering, patterns known to be design patterns describe solutions that have been developed over time to solve a recurring problem in a specific context [18]. In general, a pattern is a description of a justified solution for design problems offered in a specialised template. In this research, whenever the word 'pattern' occurs, it refers to one of design/security patterns.

Gamma et al. [18] have introduced the most popular design patterns catalogue. In their catalogue, they have described a number of well-known patterns in a template which consists of: Pattern Name, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. The authors regard the pattern solution as encapsulation of Structure, Participants, and Collaborations. Due to the ease and usefulness of the design patterns template offered by Gamma et al., many design patterns authors have employed this template when introducing new design patterns [9], [25]–[27], [1].

In this research, it is an aim to formally model the solution part of the patterns descriptions. In particular, the focus is on formalising the fully decidable statements in the solution of a pattern while taking security design patterns as an application in this research. So, whenever the phrases 'formalising a pattern', 'formally modelling a pattern', and/or 'pattern formalisation' are used, they refer to the use of Codecharts for conveying a fully decidable statement in the solution section of a security design pattern to a visual and formal Codecharts specification.

2.4 Security Patterns

Security patterns are solutions to solve recurring security problems in specific domains [9], [26]. They are aimed to combine the experience and knowledge of both security experts and software engineers to build a reusable way to solve a specific security problem. There are a growing number of catalogues [1], [9], [26]–[29] which describe security patterns. However, it has been noticed that they contain some ambiguity in their descriptions of the patterns [30].

Security patterns are claimed to be related to the well-known design patterns [18], [9]. In the description of a security pattern, there is usually a subsection which discusses the pattern relation with one or more of well-known design patterns and/or other security patterns. However, validation of such relationships has not been sufficiently covered.

A number of researchers have established their own approaches to organising and classifying security patterns [31]–[34]. These efforts were to overcome the difficulties faced by practitioners when intending to choose the appropriate pattern to apply. They were also to allow us to see the similarity and problem domain among these patterns.

Security patterns are analysed in many sources to have both structural and behavioural specifications. They have been described using UML notations and diagrams. This has a number of drawbacks such as the difficulty or impossibility of automating the verification process. From this emerges the need for using a formal design description language in order to describe the security patterns for further investigation.

Security patterns are described using a template [9]. This template consists of a number of sections as follows: Context, Problem, Solution, Consequences and See

Also. The Context section describes the situations of the problem occurrence. The Problem section describes the security problem itself. The proposed solution of the problem will be detailed in the section Solution. The solution, depending on the problem, might be at more than one level. A discussion of the advantages and the disadvantages of the solution is in the Consequences section. Finally, the See Also section will refer to other patterns when necessary. This is because introducing a solution might introduce new problems which need to be handled by other patterns. These are the main characteristics of security patterns. However, there are other characteristics but they are not specific for security patterns in comparison with the aforementioned ones.

Security patterns acquired due attention by researchers. Currently, the number of security patterns as well as the research on these patterns is increasing. As a result, many catalogues, papers and technical reports have been published to cover different aspects of them. It is difficult to list all of these resources. However, some of the most prominent works will be highlighted and discussed.

Yoder and Barcalow catalogue

J. Yoder and J. Barcalow [1] have made a cornerstone catalogue for security patterns. The authors discuss seven security patterns and then attempt to provide a suggestion of a framework which represents them in a correlation. The paper introduces and follows a template for describing a pattern which has been later used as a common pattern description template by other security patterns catalogues. However, ambiguity in the descriptions is obvious. For instance, Check Point has been described ambiguously. Moreover, late checking is to be handled by a secondary interface; however, the questions of "How" and "What" emerge.

Kienzle et al. catalogue

Kienzle et al. [28] try to create a security pattern repository with more than 26 patterns. Mainly, they describe the patterns in favour of internet applications. They also provide an excessive amount of information to suit the security needs in internet applications. However, the repository suffers from ambiguity. For example, in the Network Address Blacklist pattern description, it is stated that "client" is an element of the pattern. This is far too imprecise as the client is usually used for registered and known actors to the system. Another example is how the Blacklist controls and configures the Blocking Mechanism. Is this by instantiating new Blocking Mechanism and attaching the network address to it? Or is it by calling a method in the Blocking Mechanism which re-configures it?

Wassermann and Cheng catalogue

Wassermann and Cheng [27] aim to describe Security Patterns in a way that improves comprehensibility. The catalogue highlights a revision to most of the well-known security patterns. When investigating modelling a Checkpoint Pattern using Codechart, this this catalogue played a major role as it had the clearest description of a Checkpoint Pattern. Some other catalogues use this paper in order to reference their description of some security patterns.

Schumacher el al. catalogue

The most dominant book that concentrates on security patterns is written by Schumacher et al. [9]. The interesting point in this book is that it describes security patterns in a way that reflects securing a military base. In addition, it engages the related pattern in the description of a pattern at hand. However, despite the aforementioned advantages, the book’s descriptions and UML diagrams of the patterns suffer from ambiguity. This ambiguity will be highlighted further in the results and discussion section.

Alur et al. catalogue

Alur et al. [26] have introduced their security patterns catalogue. In this catalogue, the authors have presented more than 20 security patterns. The catalogue shows the best practices of security patterns. In addition, it describes the patterns which aim to secure applications and web services developed in J2EE. The catalogue follows the common template of pattern deception, which uses informal statements and UML diagrams. Although the catalogue does not formally describe the patterns, it presents the patterns with less ambiguity with comparison to other catalogues.

In order to compare the above discussed catalogues, it is a good idea to start with the elements of the template used for patterns descriptions. The aforementioned catalogues have a section, in the template, which lists all other names of a pattern, except the Alur et al. and Kienzle et al. catalogues. In addition, the catalogues, but Wassermann et al. catalogue, explain the problem which a pattern is designed to tackle. However, Wassermann et al and Yoder et al. catalogues are the only catalogues which motivate a pattern in the description. This might be due to that Wassermann et al catalogue improves on Yoder et al. one. In addition, all the catalogues provide a section named “Related Patterns” which refer to some relations exist between a pattern and other patterns. However, they differ in explaining the relations as some catalogues are just listing the related patterns such as Kienzle et al. catalogue.

Further differences among catalogues are the use of visual notations as Yoder et al. and Kienzle et al. catalogues are using symbolic notations (computer, actor, boxes etc..) beside the textual description in the “Solution” of a pattern description, whereas, the other catalogues uses UML class and sequence diagrams. Among Wassermann et al., Alur et al., and Schumacher el al. catalogues which use UML diagrams, Alur et al. catalogue has the most clearer and comprehensive UML diagrams. Furthermore, Alur et al. catalogue is the only catalogue that provide a sample code in a pattern description.

Some researchers tend to investigate the classifications of security patterns. Others aim to improve the templates used to describe them. For example, Alvi et al [31] have proposed a classification scheme which depends on the security flaws to organise the security patterns. Another paper [32] proposes a new representation called "Pattern Graph" to enhance the descriptions of security patterns in terms of precision.

Hafiz et al. [33] attempt to investigate the current security patterns classification and organisation approaches to reach a technique that overcomes the drawbacks in the current approaches. The authors propose a classification scheme which uses a hierarchy so that the patterns are displayed as nodes in this hierarchy. This is based on 14 security patterns to be in the hierarchy. As a result, the pattern users can navigate through and choose what fits most.

In conclusion, security problems can be at an expensive cost if not considered during the software development process. Therefore, many have proposed software security as a major part of the software development lifecycle [35]. Security patterns play a key role in software security by offering security expertise in an organised template.

Currently, security patterns are holding many researchers' interests in terms of evolution, classification, modelling, formalisation, verification and detection. However, this field of research is not covered sufficiently. It is important to mention here that, in the case of formalisation, verification and detection, security patterns solutions are referred to as security patterns.

2.5 First order predicate logic (FOPL)

In order to express declarative statements, which propositional logic could not help with, First Order Logic (FOL) was introduced which is a richer logical language. In FOL, variables and quantifiers are features which make FOL richer than propositional logic. So, a formula in FOL is a set of terms (variables, constants, and functions with finite arguments) and predicates with finite arguments [36], [37].

This topic is highlighted in this research background as some FOL formulas might be shown in some parts of this research. This is due to the fact that Codecharts is a visual and formal specification in LePUS3 [38], [39] which in turn has been defined using first order logic predicate calculus. Therefore, whenever a Codechart is shown in this research, it is understandable that the Codechart is a set of first order predicates and can be automatically translated, using TTP Toolkit [13], from visual representation to LePUS3 (first order logic predicates) and Class-Z.

2.6 Codecharts (LePUS3) and Class-Z

2.6.1 LePUS3

Codecharts are the statement of static design encoded in OOP languages. The language of Codecharts is LePUS3, which is an object-oriented design description language. LePUS3 is built on top of the strength of other specification and modelling languages. Mainly, LePUS3 came to address the following concerns [39]:

1) Rigour;
2) Scalability;
3) Minimality;
4) Program Visualization; and

5) Automated Verifiability.

LePUS3 offers solutions for the aforementioned concerns. For example, LePUS3 specification (Codecharts) is graphical. This addresses one of the concerns, namely Program Visualization. In addition, LePUS3 uses first order predicates calculus to formalise each element in the specification. This makes it a formal (rigour) graphical language. Having such formal specifications, the design conformance verification process can be automated in a more accurate and confident way. This addresses the concern of Automated Verifiability [38].

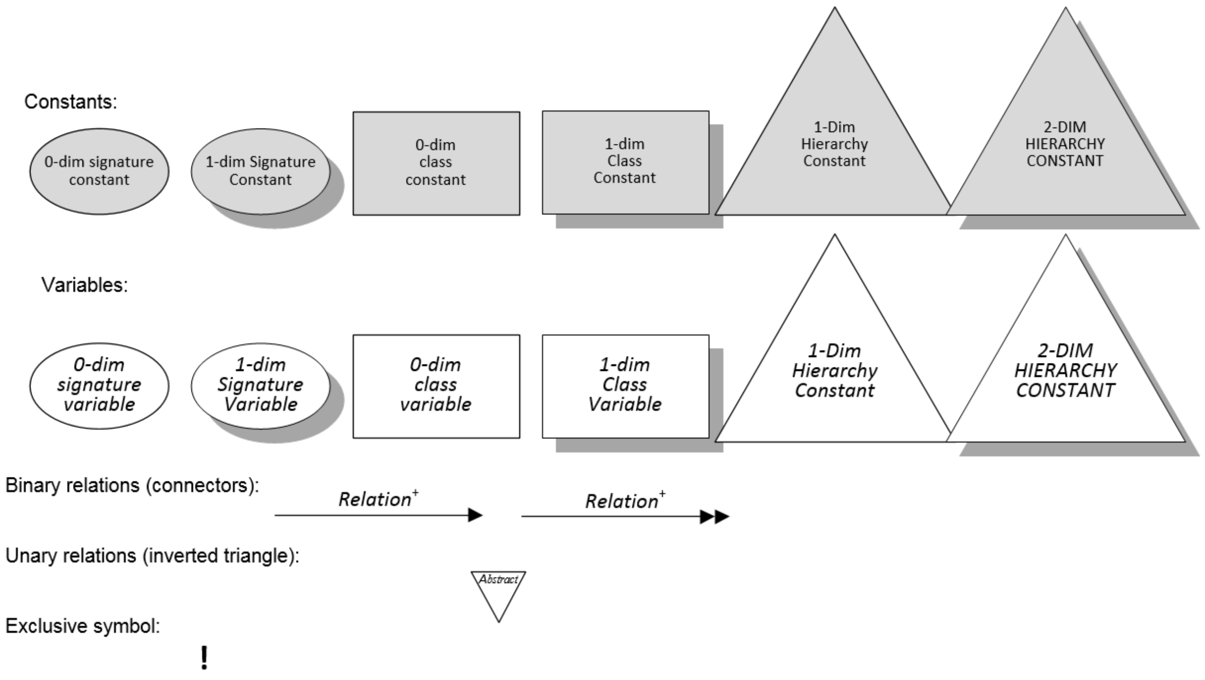


Figure 5: LePUS3 vocabulary [38]

However, this does not mean that this comes at the expense of minimality and elegance, which are LePUS3 concerns. Figure 5 shows a summary of LePUS3 visual vocabulary. In addition, LePUS3 provides means of variables in order to represent design motifs. This offers an opportunity to specify the static structure without constraints of naming. Furthermore, LePUS3 introduces an abstraction mechanism which is highly beneficial when modelling large scale software (scalability).

Abstraction mechanism relies on the notion of dimension, hierarchy and transitive relations [38]. Moreover, LePUS3 has a tool support which is called Two Tier Programming toolkit (TTP toolkit) [13]. In addition, with comparison to UML, Codecharts could outperform UML in formalising security design pattern as will be seen later in this thesis in Chapter 4.

2.6.2 LePUS3 formalism

LePUS3 is a formal object-oriented specification language. It relies on a subset of first order logic. LePUS3 is a visual and symbolic language. Its diagrams consist of terms and constraints on those terms [38], [40]. The terms and constraints are summarised and illustrated in a hierarchy form in Figure 4.

1. Terms (t)
    a. Variables (v)
        i. d-dimensional class $\mathcal{P}^d$ $\mathbb{CLASS}$
        ii. d-dimensional signature $\mathcal{P}^d$ $\mathbb{SIGNATURE}$
        iii. d-dimensional hierarchy $\mathcal{P}^d$ $\mathbb{HIERARCHY}$
    b. Constants (**<u>c</u>**)
        i. d-dimensional class $\mathcal{P}^d$ $\mathbb{CLASS}$
        ii. d-dimensional signature $\mathcal{P}^d$ $\mathbb{SIGNATURE}$
        iii. d-dimensional hierarchy $\mathcal{P}^d$ $\mathbb{HIERARCHY}$
2. Constraints (f)
    a. Relations $\mathbb{R}$
        i. BinaryRelation $BinaryRelation(t1, t2)$
            1. Inherit $Inherit\ (t1, t2)$
            2. Create $Create(t1, t2)$

3. Produce $Produce(t1, t2)$
4. Return $Return(t1, t2)$
5. Call $Call(t1, t2)$
6. Forward $Forward(t1, t2)$
7. Member $Member(t1, t2)$
8. Aggregate $Aggregate\ (t1, t2)$
9. SignatureOf $SignatureOf\ \ (t1, t2)$

ii. UnaryRelation $UnaryRelation(t)$

1. Abstract $Abstract\ (t)$
2. Method $Method\ (t)$
3. Class $Class\ (t)$
4. Signature $Signature\ (t)$

b. Predicates $\mathbb{P}$

i. Isomorphic $ISOMORPHIC(BinaryRelation, t1, t2)$

ii. Total $TOTAL\ (BinaryRelation, t1, t2)$

iii. All $ALL\ (UnaryRelation, t)$

c. Operators

i. Superimposition ($\otimes$)

$s \otimes c \quad if\ \ \langle s, c\rangle\ \in$ <u>*SignatureOf*</u> and $\langle c, s\rangle\ \in$ <u>*Member*</u>

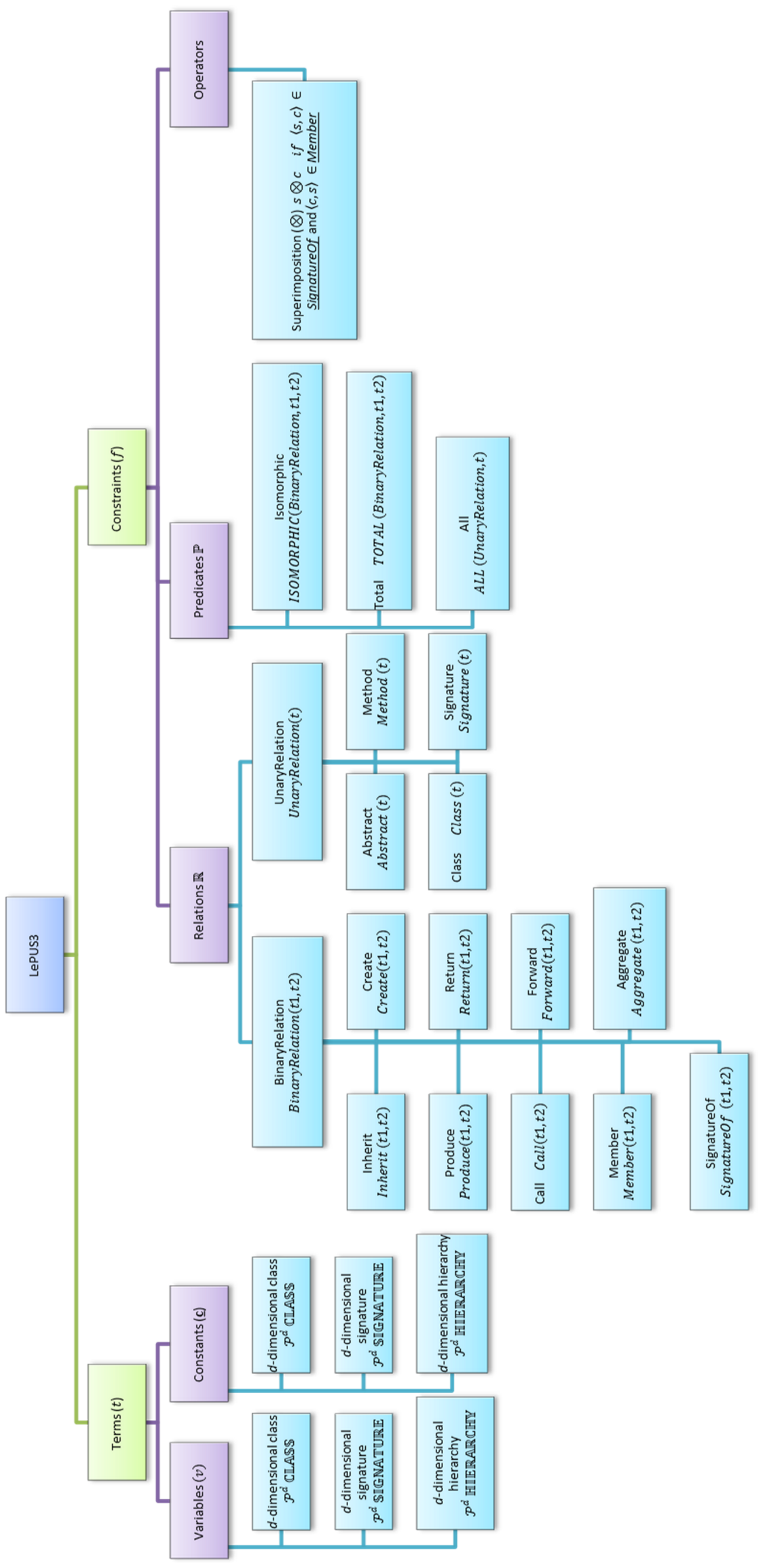

LePUS3
Terms (t)
Constraints (f)
Variables (v)
Constants (ς)
Relations $\mathbb{R}$
Predicates $\mathbb{P}$
Operators
d-dimensional class $\mathcal{P}^d$ CLASS
d-dimensional signature $\mathcal{P}^d$ SIGNATURE
d-dimensional hierarchy $\mathcal{P}^d$ HIERARCHY
d-dimensional class $\mathcal{P}^d$ CLASS
d-dimensional signature $\mathcal{P}^d$ SIGNATURE
d-dimensional hierarchy $\mathcal{P}^d$ HIERARCHY
BinaryRelation $BinaryRelation(t1,t2)$
UnaryRelation $UnaryRelation(t)$
Inherit $Inherit(t1,t2)$
Create $Create(t1,t2)$
Produce $Produce(t1,t2)$
Return $Return(t1,t2)$
Call $Call(t1,t2)$
Forward $Forward(t1,t2)$
Member $Member(t1,t2)$
Aggregate $Aggregate(t1,t2)$
SignatureOf $SignatureOf(t1,t2)$
Abstract $Abstract(t)$
Method $Method(t)$
Class $Class(t)$
Signature $Signature(t)$
Isomorphic $ISOMORPHIC(BinaryRelation,t1,t2)$
Total $TOTAL(BinaryRelation,t1,t2)$
All $ALL(UnaryRelation,t)$
Superimposition $(\otimes)$ $s \otimes c$ if $(s,c) \in \underline{SignatureOf}$ and $(c,s) \in \underline{Member}$

Figure 6: LePUS3 formalism

2.6.3 Class-Z

Class-Z is a specification language derived from Z specification language. However, unlike the Z language, which is more expressive and not limited to specific domains or purposes, it is limited to terms declarations and formulas corresponding to LePUS3 vocabulary. Class-Z is an equivalent specification language to LePUS3, however, it is used with LePUS3 as a helpful approach to resemble the formulas of LePUS3 in their traditional way. It is important to emphasise her that the formal foundation of LePUS3 is First Order Predicate Logic and all the specifications in LePUS3 (any Codecharts) are predicates in this logic, and because of the aforementioned reason, Class-Z has been used with the LePUS3, so that LePUS3 improves on the benefits of visual notation. , the two specifications (Class-Z and LePUS3) has been used together for attracting traditional audiences.

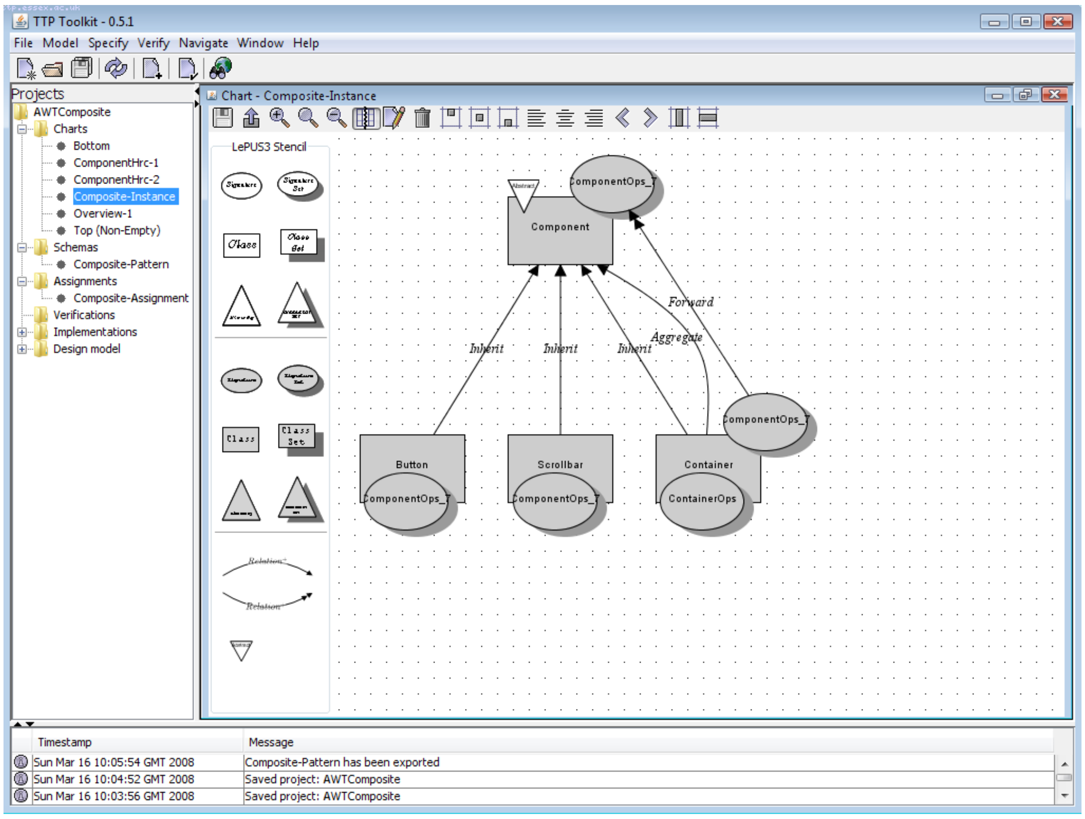
In Class-Z, specifications are in a schema form which is divided into a declarations section and a formulas section. The declarations section shows constant and variables terms, whereas the formulas section shows constraints or formulas in LePUS3 [38], [41]. Automatic converting of Codecharts (specifications in LePUS3) to Class-Z schema is one of the TTP Toolkit [13] features. An example of LePUS3 and Class-Z schema is shown in Figure 7.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Factory Method}} \\
\hline
\mathit{Factories}, \mathit{Products} : \mathbb{HIERARCHY} \\
\mathit{factoryMethod} : \mathbb{SIGNATURE} \\
\hline
\mathit{ISOMORPHIC}(\mathit{Produce}, \mathit{factoryMethod} \otimes \mathit{Factories}, \mathit{Products}) \\
\hline
\end{array}$$

Figure 7: Class-Z schema for Factory Method pattern Codechart

2.7 Two-Tier Programming Toolkit (TTP Toolkit)

The GUI prototype tool provides a number of round-trip software engineering actions. These actions start with a reverse-engineering object-oriented source code to Codecharts diagrams, which are formal specifications in LePUS3. In addition, the TTP Toolkit offers verifying design conformance of a code against Codecharts diagrams. Furthermore, it allows visualising the reverse-engineered source code, showing the software in different abstraction levels. Moreover, it has the functional feature of navigating through the software visualisation from the highest level of abstraction to a detailed and concrete view of the software. Figure 8 displays the TTP toolkit in action [13].



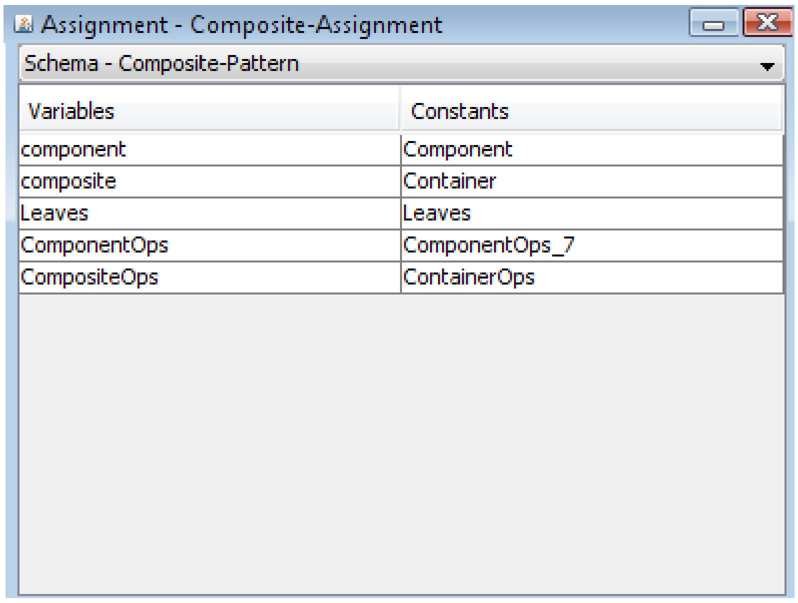
Timestamp	Message
Sun Mar 16 10:05:54 GMT 2008	Composite-Pattern has been exported
Sun Mar 16 10:04:52 GMT 2008	Saved project: AWTComposite
Sun Mar 16 10:03:56 GMT 2008	Saved project: AWTComposite

Figure 8: TTP Toolkit in action [13]

Figure 8 displays the TTP Toolkit in action. The main interface of the TTP Toolkit is divided into 4 sections namely *Menus and Quick Buttons*, *Projects*, *Workspace*, and *Information* section. The *Menus and Quick Buttons* section encapsulates all the functionalities which TTP Toolkit offers. Starting with the *File* menu, it allows to create, open, and/or save ‘ttp’ projects which are the files that store the model of analysed source code, Codecharts, assignments and verifications objects, and schemas. The *Model* menu allows a user to select any java source and analyse it to have its model. Creating new specification (Codecharts), Schema, and/or assignment, can be carried out through the *Specify* menu which also provide a set of already specified design patterns that are modelled in [38]. It is obvious that *Verify* menu provides means of creating new verification objects that TTP Toolkit verifier can work on. The *Navigate* menu offers a sophisticated function of viewing the model of the source code in different level of abstraction based on the Codecharts notations. Finally, the *Quick Buttons* are actually a way of fast access to the functionalities offers in the menus.

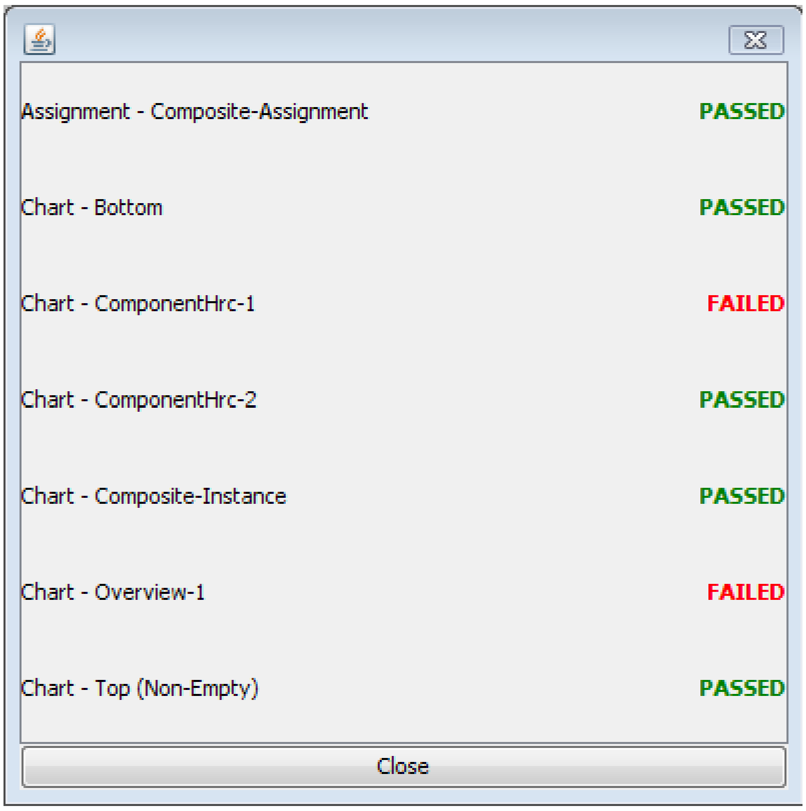
The *Project* section offers a tree viewing of the opened 'ttp' projects and the items created in them. So, opening a specification (Codecharts) in order to modify and/or delete can be done through this section. Whereas, the *Workspace* section (in the middle) allows the user to drag and drop the Codecharts notations and create the relations among them. As seen in Figure 8 the workspace is showing the modelling of Composite Pattern. Finally, the *Information* section (bottom of the window) shows the results of verifications, notifications and logging information of the projects saving, exporting, importing code analysing, etc.

The current version of the TTP toolkit provides a means of automated design conformance verification. This is carried out using a means of assignment (shown in Figure 9) which associates each variable in generic specifications with a constant (representing specific elements of concrete programs) of the same type and dimension. The verification algorithm has been automated in the TTP toolkit in a component called Verifier, which proves (or refutes) conformance of a code to your design, which is a LePUS3 specification or, in another word, a Codechart. The result of the verification is displayed using GUI as appears in Figure 10 [13]. TTP Toolkit is available for free at ttp.essex.ac.uk.



Variables	Constants
component	Component
composite	Container
Leaves	Leaves
ComponentOps	ComponentOps_7
CompositeOps	ContainerOps

Figure 9: Editing an assignment [13]



Assignment - Composite-Assignment	PASSED
Chart - Bottom	PASSED
Chart - ComponentHrc-1	FAILED
Chart - ComponentHrc-2	PASSED
Chart - Composite-Instance	PASSED
Chart - Overview-1	FAILED
Chart - Top (Non-Empty)	PASSED

Figure 10: The result of verification [13]

However, the TTP Toolkit has a number of limitations which can be overcome with further investigation of it and of LePUS3. For instance, the TTP Toolkit does not offer features for runtime-verification. The reason behind this is that it supports LePUS3 languages that only accommodate specifying decidable design statements, which are only statically checked by TTP Toolkit verifier. Another limitation is that the current version of the TTP Toolkit only supports analysing a Java source code. However, because the TTP Toolkit was developed for any object-oriented programing language, this limitation can also be overcome by further development of the tool. Finally, pattern detection (searching) is provided in the TTP Toolkit. However, this research aims to study the detection problem and develops an algorithm to solve it. The implementation of the algorithm will be integrated into the new version of the TTP Toolkit.

In this research, the TTP Toolkit is employed in order to accomplish a number of tasks. Firstly, it is used for specifying and visualising the formalised security design patterns, which are the application of this research. In addition, the verifications of the formalised security patterns against the claimed-to-implement source code is carried out with the TTP Toolkit verifier. Moreover, as this research targets studying and developing an algorithm for pattern detection, the algorithm will be integrated in the TTP Toolkit and will be used for verification of the existing instances of patterns.

2.8 Conformance checking

It is important to distinguish between software validation and software verification. Software validation is the process of checking that the software is doing the expected functionality, whereas software verification is the process of proving that the software is correct [42].

In this research, the use of verification in the pattern detection is to check that a source code has a correct architectural and structural implementation of a pattern. It is important to mention here that this research does not aim to check whether the found pattern implementation in the source code is functioning are required or as what the pattern should do (validation). So, when an instance of a pattern is found in the detection, the verification process shows whether the instance is a correct implementation against a source code. Therefore, distinguishing between verification and validation is necessary in this research.

Software consists of static and dynamic aspects. Software verification needs formal representations for those aspects of the software. These formal representations are produced using one of the known formal languages, such as propositional logic, First Order Logic, Hoare logic, etc. [36]. In order to prove the correctness of the static aspects of a software, the process is as follows: the aspects are formalised using one of the formal methods to have a set of specifications; then, the source code of the software is reverse-engineered using an appropriate tool for the chosen formal method to have a software model; finally, the model is checked against the specifications for satisfaction [43]. This process is usually referred to as *model checking, design conformance checking, static check/analysis, or design verification* [36], [38]. This process can be automated due to the fact that specifications for static aspects are decidable. A number of tools offer such automated design conformance checking such as TTP Toolkits [13], SPIN [44], KRATOS [45], Bandera [46], [47], Ptidej [48], etc.

On the other hand, proving the correctness of the dynamic aspects of a software relies on *Runtime Verification,* which is the process of checking the execution trace of a software against a set of specifications [49]. Checking the execution trace can be either *online* (which is done immediately whenever an event occurs) or *offline* (which is done after a certain amount of time or after software termination) [50]. Many have introduced their runtime verification approaches, for instance, LAVA [51], RV4WS [50], E-Chaser [49], etc.

This research aims to overcome the ambiguity in the security design patterns catalogues by formalisation through using Codecharts. So, it is important to check the conformance of the patterns against the software source code, in which a pattern is claimed to be implemented. Therefore, the term ‘pattern verification’ refers to the checking of the decidable specification (shown as Codecharts) of the solution of a pattern using the TTP Toolkit verifier [13], which is the available tool support for Codecharts.

Furthermore, this research aims to design and implement an algorithm for patterns detection from the source code. As a result, it is vital to reason out the detected instances of patterns. Therefore, verification of the instances is used with the proposed pattern detection. Again, the TTP Toolkit verifier [13] will be used to check the satisfaction of the detected instances of patterns against the pattern specifications shown in Codecharts.

3 Literature review

Note: some parts of this chapter were published in:

Alzahrani, Abdullah A. H., Majd Zohri Yafi, and Fawaz K. Alarfaj. 2014. ‘Some Considerations on UML Class Diagram Formalization Approaches’. *International Journal of Computer, Information, Systems and Control Engineering* 8 (5): 47 – 50.

3.1 Patterns formalisation

Formalisation could be defined as any process of conveying ambiguous statements or notions into precise ones [52]. This is usually achieved using a variety of mathematical and logical methods (i.e. first order logic FOL, higher order logic HOL, and temporal logic). In the context of pattern formalisation, it is the process of formally representing the pattern structure and behaviour.

In software engineering, particularly in design and security patterns, natural language statements and graphical notation (such as UML) are used for patterns specification purposes. However, this could lead to incompleteness and lack of precision. Therefore, formalisation could help with overcoming such issues. Furthermore, formalisation is an essential requirement for rigorous and automated analysis [41], [53].

UML accommodates a number of diagrams, for instance, Class, Sequence, and other diagrams which are used to represent structural and behavioural aspects of patterns and systems. UML suffers from informal representation [54], which means that it has no formal foundation. This has led to ambiguity and incompleteness in its diagrams. Consequently, rigorous analysis cannot be carried out when such issues exist. This includes consistency verification, traceability, and formal reasoning [55], [56]. Therefore, many researchers have introduced their approaches [55], [57]–[64] to formalise those diagrams. In fact, the formalisation of design and security patterns often involves the formalisation of the UML diagrams shown in the patterns descriptions.

In order to overcome the ambiguity in the UML diagrams shown in patterns or any system documentations, researchers have introduced many methodologies to present UML in a formal shape. These methodologies fall into two categories [56]: a) transforming UML to formal models [65]–[67], and b) providing abstract syntax and formal semantics for UML diagrams [68]–[70]. However, there has to be a trade-off between these categories as each category emphasizes certain aspects of UML formalisation. Many formal languages have been proposed in formalising UML such as Object Constraints Language (OCL), Z, Description Logic (DL), B, PVS, etc. Each of these languages has a number of properties which might not appear in others. As a result, UML formalisation has been studied and carried out differently according to the researchers' reasons for choosing the formalisation approach. As a result, choosing an approach to formalise those diagrams is a difficult / challenging task [56]. The following casts light on some of the UML formalisation approaches which are based on different formal languages. The focus will be on the formalisation approaches of the UML structure diagrams (Class diagram).

3.1.1 Formalization using OCL

OCL is regarded as a formal language used with UML in order to specify constraints and conditions on an UML model. It plays an important role in improving precision of the specification of UML [71], [72]. Many researches [57], [63] have used OCL to express UML syntax and semantics.

In order to explain how OCL specify UML class diagrams, it is important to unpack some of the keywords and operators that are commonly used in OCL specifications. First, there is a number of reserved words that have a meaning in OCL. For example, ***context*** is the most frequent and essential word in OCL. It is used in order to specify the context Classifier (a class in a UML diagram) of the specification expression. Another reserved words is ***def*** which specify that anything, appears after, is either a class attribute definition or a class operation definition. Using ***def*** allows specifying the attribute name, type, and the initial value. In addition, in the case of defining an operation, it allows specifying operation name, set of parameters and their types, and the returning value type.

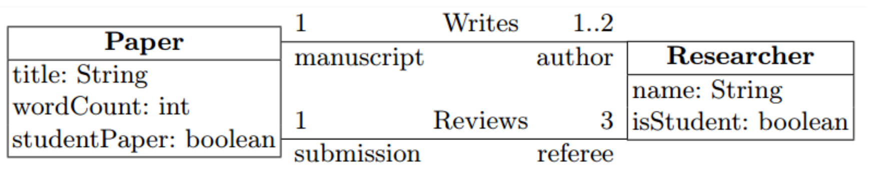
Table 1 shows an example of specifying an arbitrary class using OCL. The class name is "*Student"* which has three attributes namely "*tuitionFees"*, "*fullName"*, and "*title"* and typed *Integer*, *String*, and *String* respectively. All the attributes were initialised with values as can be seen. In addition, the class "*Student"* has one operation (method) named "*hasTitle*" and the operation takes one parameter of type *String* and returns value of type *Boolean*.

Table 1: Example of specifying an arbitrary class using OCL

Context Student
def: tuitionFees : Integer = 3000
def: fullName : String = ‘Sophie’
def: title : String = “Miss”
def: hasTitle(t : String) : Boolean = self.title->exists(title = t)

Importantly, the constraints in OCL are specified in similar manner, as the ***context*** needs to be specified, but with a use if ***inv*** keyword that abbreviates the word invariant which actually means a constraint in OCL. ***inv*** keyword specifies on classes and types and could be followed by a name of the invariant (constraint), however, it must be followed by a colon (:) to indicates the start of the constraint. Figure 11 illustrates a number of examples of invariants on two classes named “*Researcher*” and “*Paper*”.

Specifying associations among patterns is done with invariants and navigation operators which allow accessing, checking, and other operations to the class of association using the operators such as (->) specify a call relation to another operation or method call, (implies), and (.) access attribute and navigate to. Furthermore, the followings are some other keywords, OCL operations, and operators such as “implies”, “and”, “or”, “let”, “exist()”, “select()”, “forAll()”, “=”, “.”,“:”, “<>”, etc. Finally, there are many keywords, OCL operations, and operators in OCL that cannot be discussed here but it is beneficial to check them in [71].



context Researcher **inv** NoSelfReviews:
  { *Nobody can review his own paper* }
  self.submission−>excludes(self.manuscript)

context Paper **inv** PaperLength:
  { *Papers should have less than 10000 words* }
  self.wordCount < 10000

context Paper **inv** AuthorsOfStudentPaper:
  { *A paper is a student paper iff any of its authors is a student* }
  self.studentPaper = self.author−>exists(x | x.isStudent)

context Paper **inv** NoStudentReviewers:
  { *Students cannot review papers* }
  self.referee−>forAll(r | not r.isStudent)

context Paper **inv** LimitsOnStudentPapers:
  { *There must be at least one student paper and at most four* }
  Paper::allInstances()−>exists(p | p.studentPaper) and
  Paper::allInstances()−>select(p | p.studentPaper) −>size() < 5

Figure 11: Example of specifying associations between two classes using OCL from [73]

Gogolla et al. (2002) suggested an approach which [63] guides the system designers to transform UML class diagrams with constraints and relations into another UML class diagram, employing only binary associations and OCL constraints. It basically reforms the UML class diagrams to be more precise and encloses them with sets of OCL constraints. The approach stands as a de-facto in formalising UML class diagrams with OCL. Many researchers have used this approach in their work for checking design consistency [74]–[79].

Although, this approach improves the precision with the UML class diagrams, it raises the issue of losing semantic information when translating complex associations to binary ones [63, p. 93]. Furthermore, it does not support formalisation of abstraction (as there is no means of showing the class hierarchies, set of classes, and/or set of methods), which exists in patterns descriptions. In addition, formalising the security boundaries in the diagrams is lost when using this approach as this feature is not supported by the guide offered by this approach.

Another instance of using OCL to formalise the UML class diagram is demonstrated in [57] by Chavez et al. (2012). The authors address the problem of consistency between the design presented in UML and the implementation. In particular, the authors focus on formalising the composition relationship, which is often shown in the UML class diagram. The authors highlight the issue of formalising UML composition relationships in terms of lifetime and interoperability, and they suggest using OCL to formalise composition in the UML class diagram.

However, the approach concentrates on the formalisation of only one UML class diagram relationship, which is a composition relationship. Without considering other UML class diagram relationships (*dependency, aggregation ...)*, it is obvious that this approach cannot be comparable with other OCL formalisation approaches. Moreover, it does not have tool support to automate the generation of the OCL of a given class diagram as well as the verification of the consistency of a given class diagram and a given implementation.

By means of formality, OCL was founded to overcome ambiguity in UML diagrams. However, OCL itself suffers from a level of ambiguity. As a result, it does not help with formal reasoning and formal proofs [55], [80], [81]. Furthermore, it is difficult to be checked and detected, so it raises issues in the development and the maintenance process of the software system. Moreover, OCL does not seem to be sufficient when stating complex constraints [55].

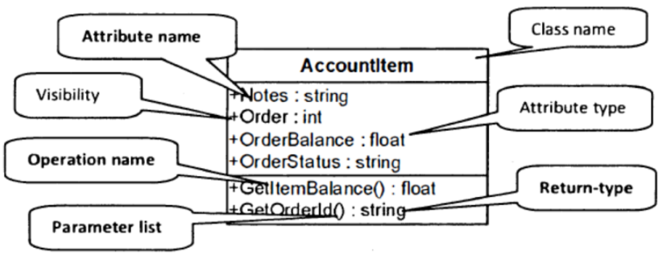
On the other hand, Codecharts allows formalising the structure of patterns or systems precisely, visually, and concisely with the feature of a small number of vocabulary. Furthermore, Codecharts supports formalisation of abstraction as it offers tokens for this, such as a hierarchy symbol, a set of method signatures symbol, and a set of classes symbol. The current version of OCL, however, does not support abstraction. Moreover, loss of information occurs when dealing with complex constraints.

In addition, Codecharts has tool support for specifying design conformance, translating it to LePUS3, and verifying it. On the other hand, OCL, to the best of my knowledge, does not have competing tool support. None of the tools, which apply formalising UML class diagrams using OCL, is available for investigation; whereas Codecharts tool support is publicly available at [ttp.essex.ac.uk].

3.1.2 Formalisation using DL

Description Logic (DL) is one of many formal languages for knowledge representation. It is built on a mathematical foundation and supports formal reasoning. A number of description logic languages exist, such as AL and ALEN [82]. Many researchers [55], [58], [61], [62], [64] have tried to formalise UML in DL.

Kadir et al. (2010) [55] have proposed an approach to formalise a UML class diagram using DL. They have assessed their approach to UML formalisation as not satisfactory. This is due to two reasons: (1) many properties, such as dependency, are not defined, and (2) the formalisation is done manually and there is no tool to automate it. Figure 12 shows the formalisation of one attribute and one operation of one class in a class diagram [55]. It can be seen that formalisation is textual and will be long in case of formalising all operations and attributes of all classes in any medium-size class diagram. It is obvious that employing such an approach in the process of the formalisation of a large-scale software can lead to an unreadable formal specification, which can consequently be error prone. Moreover, it would need a solid mathematical foundation to be understood and verified.



$$AccountItem \sqsubseteq \forall P_{GetOrderId()}.string \sqcap (\leq 1P_{GetOrderId()})$$

$$AccountItem \sqsubseteq \forall Notes.string \sqcap \exists Notes \sqcap (\leq 1Notes)$$

Figure 12: An example of formalising one attribute and one operation of a class in DL [55]

In order to explain the formalization example in Figure 12, it is beneficial to translate it in English. So, the first line of formalization means that there is a class named "*AccountItem*" which includes ($\sqsubseteq$ quantifier) an operation ($\forall P$) named "*GetOrderId*" with a return value of type (.) "*String*" and ($\sqcap$) the operation visibility is publicly visible and has one restriction ($\geq$). In the same way an attribute is formalized but without the use of operation identifier ($\forall P$).

Zhihong et al. (2003) [61] considered the formalisation of a UML class diagram in DL but from a different perspective. They considered the formalisation process itself. They first provided a summary of the comparison between DL languages used in formalising UML. Then, they addressed some concerns when choosing a DL language for formalisation. They suggested some solutions to the problems which occur in formalising a UML class diagram. They concluded that formalising UML in DL is a difficult task.

Another approach to formalising UML diagrams with DL was suggested in [62] by Calì et al. (2002). The authors have chosen the DLR description logic language in order to formalise the UML class diagram in terms of classes, associations, and constraints. They have shown how to map the constructs of a class diagram to the corresponding formalism in DL. However, they did not consider those aspects which relate to the source code (public, protected and qualifiers for methods and attributes) when formalising the class diagrams. Moreover, the approach does not support the formalisation of some advanced constraints, in particular the constraints on attributes names and operations parameters [60]. The approach was experimented in FACT [83]. Although, the work is promising, many properties need to be considered to mature this formal framework (such as modelling and reasoning out objects and links), as the authors concluded.

Some researchers chose to study the use of DL for formalisation from a different perspective. For instance, Berardi et al. (2001) [64] carried out an experimental investigation of the use of the most dominant DL-based reasoning systems to reason out UML class diagrams. The authors illustrate their approach of formalising the UML class diagram in the DLR description logic language. The approach they used in formalisation is similar to the work of Calì et al. (2002) in [62]; however, the difference is only that the work in [62] pays more attention to formalising the constraints shown in UML class diagrams. Berardi et al. [64] reported detailed results about the most popular DL-based reasoning systems, namely FACT [83] and RACER [84]. Briefly, the result of the experiment showed that the tested DL-based reasoning systems suffer from critical efficiency issues when dealing with knowledge bases.

DL is a formal approach which can be used to formalise UML class and other diagrams. Its main features are soundness and completeness, which are essential in rigorous reasoning. In addition, DL has a number of languages which vary in their features. However, DL cannot formally represent all UML properties, such as dependency relation [55], [58], [60]. In addition, when formalising a UML class diagram in DL, the choice of the DL language needs to be considered carefully [61].

Furthermore, as DL's languages are similar, transferring between DL languages can happen easily, which makes it difficult to say which DL language is used if an explicit statement does not exist. In addition, the process of formalisation using DL is a challenging task as it is still done manually and needs solid mathematical foundations and skills in DL.

On the other hand, Codecharts is a formal language, as it is built on First Order logic, which makes it superior and more expressive, since DL and DLR are regarded to be subsets of First Order logic [85]. In addition, Codecharts is visual, which makes formalisation easier in produce and understanding. Furthermore, Codecharts supports formalising abstraction and offers a number of vocabularies for abstraction; whereas the current approaches of formalisation using DL do not offer this support [55]. Supporting abstraction helps with the formalisation of large scale system models. Finally, Codecharts has a tool support for specifying design performance, verifying it automatically, and translating it automatically to LePUS3 formulas. Whereas the formalisation in DL is done manually and the current tools dealing with DL formalisation suffer from issues such as efficiency.

3.1.3 Formalization using Z language

Z is a specification language strongly typed in mathematics [59]. Since its introduction, it has been an interest for formalisation advocates. This resulted in introducing Object- Z, which is an extension of Z. Object-Z was developed to improve Z in many aspects, mainly in structuring and object-oriented representations. This is to enhance effectiveness in specifying large and medium scale software systems. However, it is claimed that a specification in Z is also a specification in Object-Z [86]. It seems a true claim as Z's syntax and semantics are parts of Object-Z, however, the latter improves in inheritance, polymorphism, and encapsulation. UML formalisation using Z and Object-Z has been of interest to many researchers.

Sengupta et al. (2008) [59], for instance, have proposed a methodology to formalise different kinds of UML diagrams in Z language and represent the result visually using an Entity-Relationship (ER) diagram. The authors have clarified their methodology of formalising a UML class, a Use-Case, and Sequence diagrams. However, the proposed approach has a number of drawbacks. First, the ER diagram can be large when representing an industry-scale system. Second, the process of the transformation of Z specification into an ER diagram is not clear. This would introduce more ambiguity than UML diagrams do. Finally, the proposed approach has not been implemented in a tool which automates formalisation into Z, ER representation, and consistency verification.

Another work has been carried out by Mostafa et al. (2007) [60] to formalise UML diagrams in Z language. The authors introduce a manual which provides formal semantics for a Use-Case diagram, a Class diagram, and a State Machine diagram in Z specification language. They have, in addition, implemented a tool to support their approach. The tool is claimed to be able to check diagrams and automatically generate the appropriate Z specification if there is no violation of constraints. Additionally, the tool could generate a source code for class diagrams in C#, visual basic and JavaScript.

Although the approach offers a straightforward formalisation guide for skilled people in Z, it does not consider formalising abstraction in UML class diagrams. Furthermore, the authors have shown no evidence of the implemented tool. They only mention, in the conclusion section, that a tool was implemented and list what it does.

ActorGeneralization

Actors

$aparent, achild: \mathbb{F}\ Name$

$\forall x, y, z: human; ageneralization: \mathbb{P}\ (aparent \times achild)$
$\quad \mid (x, y) \in ageneralization \land (y, z) \in ageneralization$
$\quad \bullet\ (x, z) \in ageneralization$
$\forall x, y, z: non\_human;$
$\quad ageneralization: \mathbb{P}\ (aparent \times achild)$
$\quad \mid (x, y) \in ageneralization \land (y, z) \in ageneralization$
$\quad \bullet\ (x, z) \in ageneralization$
$\forall p: aparent; c: achild;$
$\quad ageneralization: \mathbb{P}\ (aparent \times achild)$
$\quad \mid (p, c) \in ageneralization \bullet\ (c, p) \notin ageneralization$

Figure 13: An example of formalising one generalisation relation between two classes in Z [60]

Figure 13 demonstrates an example of a generalization relation in Z as shown in Mostafa et al. work [60]. Clearly, from the figure, a number of shortcomings of the approach are noticed such as asymmetry. In addition, this approach has the limitation when dealing with a large-scale software. The reason behind this is the overwhelming specification generated, which makes carrying out any formal reasoning a difficult process.

In conclusion, Z specification is a formal method which can help in formal reasoning and formal proofs. However, Z notations are not graphical and need a solid mathematical background to be formed and understood [59]. In addition, Z lacks some notations such as Interface. It also lacks clarity when the specification is for large scale software systems [86]. Furthermore, a glance at the state-of-the-art approaches reveals that there is no tool to support automating the formalisation of UML diagrams in Z and the detection of Z specification from a source code.

On the other hand, Codecharts is a formal graphical language. It supports formalising abstraction shown in design models. Furthermore, the TTP Toolkit [13] provides considerable tool support for Codecharts. The tool offers a number of actions that can automate verification of consistency, translating to LePUS3 formulas, and visualisation.

3.1.4 A Comparison of Formalisation Methods

After reviewing many approaches to formalisation of UML class diagrams using different methods, the following compares between the three aforementioned methods used in the approaches reviewed. Table 2 describes the main findings of the comparison. The following criteria have been used in the comparison as these criteria are either regarded as formalisation concerns or as consequences of the formalisation processes.

1. ***Full Formality***: it highlights the approaches which have formality issues.
2. ***Information loss***: it highlights the possibility of losing some information during the process of transferring the UML class diagram into a formal specification. Basically, the information loss occurs when an approach is not able to formalise some information in the class diagrams such as dependency relation or complex constraints.
3. ***Abstraction support***: it shows to what extent an approach is responsive to large-scale software representation. In addition, it displays the ability of the approach to formalise abstraction shown in a class diagram.
4. ***Complex constraints support***: it refers to whether an approach is used in formalising complex constraints.
5. ***Automation tool support***: it illustrates the availability of any tool which automates the formalisation processes.
6. ***Math background***: it describes the need for a solid mathematical background during the formalisation process or thereafter for understanding and checking the formalisation outputs.

7. ***Graphical***: it refers to the outputs of the formalisation; i.e. whether the outputs are graphical or textual.
8. ***Behaviors formalisations***: they show whether the approaches in the formalisation method is able to formalise behavior models.

Table 2: Comparison of the Formalisation approaches using different methods

Feature \ Formalization method	Z	DL	OCL	Codecharts
Fully Formal	√	√	×	√
Overcome Information loss	×	×	×	√
Modelling Abstraction support	×	×	×	√
Complex constraints support	√	√	×	√
Automation tool support	√	×	×	√
Naive Math background	×	×	√	√
Graphical	×	×	×	√
Behaviours formalizations	√	√	×	×

Formalisation approaches vary in order to fulfil the different needs. This makes each approach unique in the way it deals with formalisation of design models. The difference does not make one approach outperform the others; however, it shows the main purpose behind the introduction of such an approach. Table 2 demonstrates the outcome results from comparing a number of UML class diagram formalisation approaches in OCL, DL, and Z.

It can be seen from Table 2 that the approaches of formalization reviewed differ in their abilities to meet all the criteria. This might be due to the chosen formalisation methods (OCL, DL, or Z) or due to the concerns which were the motivations for the introduction of the formalisation approach. However, Codecharts shows an ability to meet all the criteria, except for the last one, which is modelling the behaviors. This limitation in Codecharts is due to the fact that Codecharts is only able to model fully decidable statements, which is often not the case with behavior modelling.

However, Codecharts is ***graphical,*** which means that it does not require special math skills to be understood. This leads to satisfying the criterion of no ***Math background*** is needed. It is ***formal,*** which means each graphical vocabulary has a formal meaning in First Order Logic. This leads to satisfying the criterion of ***Information loss*** as no transferring is required. In addition, Codecharts has a ***tool support*** which is shown in the TTP Toolkit [13]. Furthermore, it ***supports formalizing abstraction,*** which is shown in design models, by offering a number of vocabulary items for this purpose. Finally, Codecharts is a visual specification in LePUS3, which is formulas built on First Order Logic. As a result, one of the important features of First Order Logic is expressiveness, which allows ***supporting formalisation of complex constraints***.

3.2 Pattern detection

Design pattern detection is defined as the activity of finding design motifs in a source code of subject system. In this context, the activity is limited to find the occurrences or instances of the “Solution” proposed in the patterns descriptions. Often, the detection activity includes a number of sub processes such as pattern solution formalisation, source code reverse-engineering, occurrences searching, and occurrences reasoning/explanation [11], [87], [88], [3], [4], [89], [8], [90]–[92].

Pattern Detection is to look for instances of a given or selected pattern in a given source code. The search is divided into two classes, which are static search and dynamic search. The former search is required to perform the latter search. However, not all the existing pattern detection approaches combine the two searches. The static search is the richer and the core search as it allows finding the structure and the locations of the instances within the source code. Whereas the dynamic search allows further checks on the instances found by explaining the execution trace of the source code or by plugging some chunks on codes in the original source code in order to log on the source code components activity during the execution time [11].

The Pattern Detection problem has been the focus for many researchers [6], [87], [88], [3], [4], [89], [8], [93]–[97], [5], [7], [98], who have introduced their approaches in order to solve it. In the next section, a number of state-of-art pattern detection systems and approaches will be reviewed. The review will be focused on the definitions of the pattern detection approaches, formalisation methods, search means, and analysis of the main findings and issues.

3.2.1 SPQR

One of the early and expressive pattern detection systems is SPQR (System for Pattern Query and Recognition).. SPQR tackles POML (Pattern/Object Mark-up Language) schema as inputs and outputs. It uses Otter automated theorem prover to find the instances of pattern from the source code. SPQR aims to ease the documentation and understanding of systems from the source code [99].

The SPQR has been experimented to detect instances, from a source code, of a number of design patterns described in [18]. The experiment was conducted on a small scale source code. Furthermore, a possibility of performance issue in the case of a large-scale source code is present [8], [93], [99]. Moreover, the SPQR system is not available for testing and many researchers find that the system approach cannot be rebuilt easily [90].

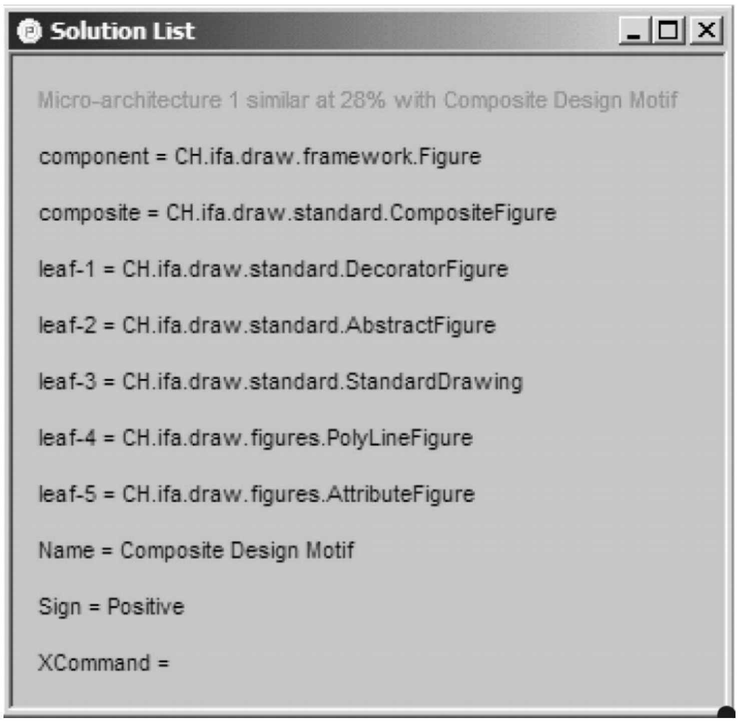
SPQR depends on the extracted structural relations to detect patterns instances. This might result in high false positive or negative ratios [100]. Moreover, SPQR has the limitation of being specific to the C++ source code. In addition, SPQR uses ρ-calculus, which does not support data flow [101].

3.2.2 Ptidej

Another example is Ptidej (Pattern Trace Identification, Detection, and Enhancement in Java). Ptidej is a system for analysing and maintaining object-oriented software systems. It is comprised of a number of sub-components as follows: a) SAD, which is a tool for detecting and correcting architecture defects; b) EPI, which is a design patterns occurrence identification tool; c) DRAM, which is a tool for visualising static and behaviour properties of a software system with matrices; d) Aspects, which models and computes matrices. As EPIis the design patterns identification tool in Ptidej, it is a relevant tool to this research and will be used when referring to Ptidej. EPI is designed to find occurrences of design patterns using a bio-informatics string matching algorithm. EPI is claimed to find exact as well as approximate pattern occurrences [7].

```
Variables:
  client
  component
  composite
  leaf
Constraints:
  association(client, component)
  inheritance(component, composite)
  inheritance(component, leaf)
  composition(composite, component)
```

Figure 14: Composite design motif in Ptidej [3]



```
Micro-architecture 1 similar at 28% with Composite Design Motif
component = CH.ifa.draw.framework.Figure
composite = CH.ifa.draw.standard.CompositeFigure
leaf-1 = CH.ifa.draw.standard.DecoratorFigure
leaf-2 = CH.ifa.draw.standard.AbstractFigure
leaf-3 = CH.ifa.draw.standard.StandardDrawing
leaf-4 = CH.ifa.draw.figures.PolyLineFigure
leaf-5 = CH.ifa.draw.figures.AttributeFigure
Name = Composite Design Motif
Sign = Positive
XCommand =
```

Figure 15: Ptidej output [7], [48], [102]

Ptidej is a large project which aims to cover many aspects such as Quality models, Program comprehension, Test-case generation, Design smells, Linguistic smells, Evolution patterns, and Features and requirements [48]. The next will concentrate on Ptidej efforts on the topics relevant to design patterns detection and identification which is the interest of this research.

Ptidej relies on an improved version of a design pattern detection system called DeMIMA [4]. Figure 14 shows the use of Constraint Programming in Ptidej in order to represent Composite pattern as input. In addition, Figure 15 depicts the Ptidej outputs of the pattern detection of Composite shown in Figure 14.

3.2.3 DeMIMA

From the source code, DeMIMA [4] aims to find microarchitectures (which are generally class diagrams) that are similar to design motifs. Constraint Programming and tree search methodologies are employed in order to find the patterns matches. DeMIMA is a semi-automatic pattern detection system. Its pattern detection algorithm tries to recognise the pattern constraints in the program model. However, it misses composition relationships. As a result, this reduces the number of exact instance of the pattern. Therefore, the algorithm does not provide all instances.

Furthermore, an early experiment has been carried out to check the approach performance, microarchitectures validation, and accuracy. The main findings show that DeMIMA still suffers from efficiency issues as it takes an average of 50 minutes to find all the microarchitectures of one design pattern. Furthermore, the accuracy is not at the desired level [4], [96].

An evolution of DeMIMA is presented in [3]. The improvement was towards efficiency and accuracy. The authors adopted numerical signatures in order to enhance the previous work on design pattern identification. In the improvement, Constraint programming is still the methodology applied. However, the performance is still an issue when dealing with a large-scale software. For instance, the authors conducted an experiment on JUnit, which has 236 classes and 1,995 methods, to identify the occurrences of Adapter, Abstract factory and Composite patterns. The results show that the identification of the Adapter and Composite patterns is still inefficient, whereas the identification of Abstract factory has significantly improved and all instances of this pattern were found quicker with comparison to other aforementioned patterns. Furthermore, the computation of the numerical signatures introduces more overhead for the identification process. Moreover, the accuracy was not improved significantly as the author concentrated on the recall more than the precision [96].

In conclusion, a number of well-known pattern detection/identification systems have been reviewed. Those systems use different approaches for patterns representation as well as different patterns finding algorithms. Each system has its strengths and drawbacks. Form what has been done and what is missing, this research aims to fill the research gap.

Form the above review, it can be seen clearly that the area of pattern detection still suffers from the problem of inefficiency. The aforementioned state-of-art pattern detection systems can be evidence for such a claim. Furthermore, as SPQR is limited to the C++ source code and Ptidej is limited to the Java source code, the need for a more general pattern detection system emerges. This means a system which can detect design patterns in any platforms of source codes (e.g. Java, C++, C#, Python,…..).

Additionally, improving system understanding, maintainability, and evolution are the main goals for pattern detection systems, and in order to achieve those goals, a fully automated pattern detection system is required. The state-of-art pattern detection systems are semi-automated as they need human interaction at some point in the detection process [7], [99].

Furthermore, the state-of-art pattern detection systems are limited to specific patterns to be detected. Usually, these patterns are hard coded and known in advance. For instance, SPQR uses a component called OTTER, which finds instances of patterns from a fixed and hard coded collection called Elemental Design Patterns (EDPs). An example is shown in Figure 16. The OTTER finds instances using inference, based on certain hard coded rules, and then provides proofs which will be analysed by the proof2pattern component to make the final outputs in the form of ObjectXML as shown in Figure 17 [98].

As a result, SPQR takes a source code as an input and uses a fixed set of hard coded patterns to find only a first instance of the chosen pattern. This finally generates the result in a form of ObjectXML. It can be concluded form this that SPQR does not find all the instances and is not a general pattern detection system as it relies on a fixed hard coded pattern to be looked for. On the other hand, this research aims to produce a general pattern detection system which is able to find all instances on any given pattern in any given implementation. Moreover, it aims to visualise the output to improve the readability and understanding of the result.

$$\frac{\begin{array}{l}\textbf{ObjectRecursion}(Component, Decorator_i{}^{i\in 1\ldots m},\\ \quad ConcreteComponent_j{}^{j\in 1\ldots n}, \textbf{any}),\\ \textbf{ExtendMethod}(Decorator,\\ \quad ConcreteDecoratorB_k{}^{k\in 1\ldots o}, operation_k^{k\in 1\ldots o}),\end{array}}{\begin{array}{l}\textbf{Decorator}(Component, Decorator_i{}^{i\in 1\ldots m},\\ \quad ConcreteComponent_j{}^{j\in 1\ldots n},\\ \quad ConcreteDecoratorB_k{}^{k\in 1\ldots o},\\ \quad ConcreteDecoratorA_l{}^{l\in 1\ldots p},\\ \quad operation_k^{k\in 1\ldots o+p})\end{array}}$$

Figure 16: An example of EDP [98]

```
<pattern name="Decorator">
    <role name="Component"> "File" </role>
    <role name="Decorator"> "FilePile" </role>
    <role name="ConcreteComponent">
        "FileFAT" </role>
    <role name="ConcreteDecorator">
        "FilePileFixed" </role>
    <role name="operation"> "op1" </role>
</pattern>
```

Figure 17: SPQR output [98]

Although many studies have been carried out to improve the Ptidej and DeMIMA pattern detection systems, limitations and issues are still present. For example, Ptidej and DeMIMA offer a fixed set of design patterns which can be looked for in any reverse-engineered source code. The design patterns are described in the form of a Constraint Satisfaction Problem (CSP). As a result, Ptidej and DeMIMA have a fixed and hard coded specialised constraints library to describe the design patterns [3].

So, again, these pattern detection systems are restricted to a hard coded design patterns. Furthermore, the outputs of Ptidej detection systems are not all the exact instances of a chosen pattern. In fact, the result of the detection process includes a considerable number of similar patterns. Finally, the result is shown in textual format. Clearly, this is unlike the general pattern detection system which this research aims to accomplish.

The final point is that the current pattern detection systems represent patterns either in a new or in a special language to transform the patterns from their UML diagrams. In the case of using a new language, this has the shortcoming of being textual and not graphical. Also, this requires the user of such a system to study this language. This leads to making such systems as not easy-to-use systems. In the case of UML transformation, a considerable chance of lost information is present [61]. As a result, this research aims to overcome the issues discussed above in pattern detection systems by proposing a system which is efficient, easy-to-use, and fully automated. Furthermore, it should rely on a formal, graphical, and easy-to-understand pattern description language.

3.2.4 Other detection approaches

3.2.4.1 Tsantalis et al. approach

Tsantalis et al. (2006) [8] have proposed a new pattern detection approach which uses similarity scoring between graph vertices in order to find the pattern instances. In this approach each pattern's relation (generalisation, association, …..) are shown as a matrix of size $n \times n$ where n is the number of classes in the pattern. The same applies to the program representation. As a result, the pattern as well as the source code are represented as adjacency Matrices. So, if we have a pattern *P* that has 3 classes A, B, and C, and the class A and C are in generalisation relation which means that C inherit from A, the adjust matric for this relation as follows:

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \end{array}$$

A pattern and the software system are divided into subsystems matrices. A subsystem is a set of components or pattern participants that form a hierarchy structure. Next, a specialised algorithm is used to calculate similarity scoring between vertices of all matrices for the pattern with the corresponding ones for the source code in order to find the pattern instances.

The authors have tested their approach on three large scale software systems, namely JHotDraw, JRefactory, and JUnit. The patterns used in their experiment are: Adapter/Command State/Strategy, Composite, Decorator, Visitor, Observer, and Prototype. The approach provides a very efficient way for pattern detection. Furthermore, unlike other pattern detection approaches and systems, the amount of memory needed for the pattern detection is at the minimum.

Although the approach offers a very efficient methodology for design patterns detection, it is limited to a fixed, predefined, and hard coded set of patterns to be detected. In addition, as the approach relies on similarity scoring where the score between matrices bounds in the range of 0 and 1, it cannot find all exact instances. Rather, it indicates the instances with higher similarity scores. Moreover, the construction of subsystems might introduce invalid or unnecessary candidates, especially when a subsystem is linked to two or more hierarchy roots. Furthermore, the approach does not provide or use any reasoning methods on the detected instances. In addition, as the authors claim that the approach has been implemented, it was not possible to find a tool which implements the approach in order to tests the approach with other patterns and software source codes.

On the other hand, in this research aims to offer a methodology for a general pattern detection. So, the user can detect any design or user-defined patterns. Moreover, specifying the pattern (Codecharts) is rather easier as it is visual but formal. In addition, the algorithm proposed in this research is designed to find all exact instances of a given pattern in a given source code. This will be integrated with the TTP Toolkit (a tool support for Codecharts) which will be used as well for reasoning and verifying the pattern instances detected by our algorithm. As this research aims to integrate the pattern detection approach in the TTP Toolkit, it will be available for testing and further investigations on the TTP Toolkit at ttp.essex.ac.uk.

3.2.4.2 Dong et al. approach (DP-Miner)

Dong et al. (2007) [5] have introduced their approach for pattern detection named Design Pattern Miner (DP-Miner). The approach is divided into three phases: structural analysis, behavioural analysis, and semantic analysis. The approach uses a manner of matrices, weight, and type in order to represent a pattern. In addition, it uses the same manner for representing the reverse-engineered software system source code. With such representation, the authors propose an algorithm to calculate the similarity and find the pattern instances statically for further behavioural analysis.

The DP-Miner pattern detection system was developed on this approach. In addition, the authors conducted a case study in order to test and evaluate the approach and the DP-miner tool support. The case study was to find a number design pattern (namely: Adapter, Composite, Strategy, and Bridge patterns) in a large scale open source code, which was Java.awt package in JDK 1_4. The detected instances are shown in the textual form as a table and in the visual form as a class diagram.

Although DP-Miner seems to be an efficient, user-friendly, and complete pattern detection system, it has a number of limitations and issues. First of all, it is, like other pattern detection systems, only limited to a fixed, pre-defined, and hard coded set of design patterns to be detected. Currently, it is limited to only 4 design patterns, namely Adapter, Composite, Strategy, and Bridge patterns. On the other hand, as mentioned before, this research aims to offer a general pattern detection approach for detecting design, security, and user-defined patterns; whereas this could overflow in the DP-Miner as the optimisation of it relies on the attributes number, which might be large for a general pattern detection system.

In addition, DP-Miner has no reasoning in the detected pattern instances. Moreover, the correct instances are manually checked in the source code by an expert human interaction. This makes it a semi-automatic pattern detection. However, this research pattern detection approach, as mentioned before, will be integrated in the TTP Toolkit and will be using the TTP Toolkit verifier to check the instances and reasoning in them.

Finally, DP-Miner employs in the search algorithm the semantics which are shown in source code design documentation, inline codes comments, methods and classes naming. It is obvious that if a correct, clear, and up-to-date documentation of the source code is available, then there is no need for pattern detection. In addition, inline code comments are often more misleading for humans than pattern detection. Therefore, relying on these might be misleading and might cause the elimination of some correct instances. This dependency on the semantics introduces the possibility that DP-miner might miss or drop some correct instances. Therefore, it cannot be said that DP-Miner finds all the pattern instances, unlike the pattern detection approach in this research which aims to find all instances.

3.2.4.3 Uchiyama et al. approach

A new approach was introduced by Uchiyama et al. (2014) [94] in order to detect design patterns from an object-oriented source code. The approach aims to solve the problem of detecting patterns with same class structure. In addition, it aims to solve the problem of finding a pattern when pattern applications are different. Furthermore, the approach is limited to the Java source code.

In order to solve the above mentioned problems, the approach employs machine learning and source code matrices. The approach is developed to solve the problem of pattern detection automatically. However, human interaction is essential in the role judgment phase as well as some other steps.

In this approach, the pattern is described using the GQM method (Goal Question Metric). However, this method has a problem in effectiveness and ease-of-use [103]. Moreover, describing a pattern using such a method requires patterns' specialists and experience. Additionally, beside the fact that the output is textual descriptions, the problem of lack of Questions might occur. This leads to unsatisfactory results in the approach phases as well as a loss of patterns information. In this research, using Codecharts offers a visual and formal method to describe the decidable descriptions of a pattern. Furthermore, the tool support TTP Toolkit [13] offers user friendly interface to describe a pattern.

The patterns to be detected in Uchiyama et al. [94] approach are only 5 hard-coded and fixed sets of patterns, in contrast, this research overcomes this limitation as it offers an approach to detect any well-known design patterns modelled using Codecharts. Moreover, this research facilitate detecting any user defined patterns.

Therefore, pattern detection in this research can be regarded as a general pattern detection approach.

The detection process in current pattern detection approaches relies on relations among components in patterns and on relations among source code components. In the approach proposed by [94], they only consider a limited set of relations which are only 4 relations, namely inheritance, interface, implementation, and aggregation. Unlike this research which aims to consider more relations by using Codecharts in order to employ them to gain more accurate and specified results. This research considers the following relations: Call, Return, Forward, Produce, Create, Member, and others beside the relations mentioned in the compared-with approach.

Finally, in this research, a specialised algorithm has been developed to detect patterns instances from the Java source code or any object-oriented source code that can be model-extracted using TTP Toolkit. The algorithm finds all instances and locate them in the source code in a fully automated process, whereas the [94] approach relies on manual human judgement to decide the most likely to be a class playing a role in a pattern. Moreover, it does not find all instances of a pattern compared with the pattern detection in this research; in fact, it ranks classes from a source code according to its relevance to the role of each participant in a pattern.

3.2.4.4 Mirakhorli et al. approach

Mirakhorli et al. (2012) [104] have introduced their approach for detecting tactics. The authors distinguish clearly between a tactic and a design pattern as they refer to the former to be a group of classes and their association, and the latter to be a group of roles and interactions. The approach aims to solve the problem of automating the construction of traceability links for architectural tactics. The solution proposed in this approach relies on machine learning, information retrieval, and structural analysis.

As mentioned, Mirakhorli et al [104] approach defines a tactic in terms of roles and interactions. However, the authors divide the roles into two categories: primary roles and additional roles. In their approach, they consider only the primary roles to be detected, where there are usually two roles in each tactic considered in their work.

The tactics in Mirakhorli et al [104] approach are described using tTIMs (Tactic Traceability Information Models) in order to describe the design decisions [105]. tTIMs provides infrastructure to visualise software architecture decisions using UML-like diagrams. Although the tTIMs is a great method to help in understanding the software architecture, tTIMs is not formally defined. In this research, design obligations are described using a formal and visual method, which is Codecharts. Formality is essential when rigorous reasoning about the results is needed. Consequently, in this research you use the TTP Toolkit automatic verifier [13] in order to verify the output of the pattern detection approach of this research, whereas in Mirakhorli et al approach, the verification process is done all manually and with the need of experts to carry it out.

After describing the design decisions, the approach of Mirakhorli et al needs a human interaction in order to link the architectural elements to proxies in the source code. This interaction needs to be with the developer or the analyst. The mapping is important in order to have traceability information inherited in tTIMs; however, the mapping is done manually, which makes this approach a semi-automatic approach.

This research aims to find all instances of a given design pattern in a given source code. This makes the pattern detection, of this research, surpasses Mirakhorli et al approach which uses Information retrieval to detect instances of design decisions and Information retrieval depends on terminology, so some instances might be missed. Furthermore, the authors [104] highlight the risks of the correctness of the training and test sets as well as the time efficiency. Finally, in this research, in order to detect all instances from a given source code, the detection relies on design decisions showed as a visual and formal input (Codecharts). However, beside tTIMs of the architectural decisions, Mirakhorli et al approach requires a source code and its documentations in advance in order to pre-analyse the result and enhance the results when needed.

4 Security Patterns Formalisation

Note: some parts of this chapter were published in:

Alzahrani, Abdullah A. H., and Amnon Eden. 2013. ‘UML versus LePUS3 for Modelling Structure of Security Patterns’. In, 260–69. Kyiv, UA: Cybernetics Faculty of Taras Shevchenko National University of Kyiv.

Design patterns formalisation is the process of having the pattern in a formal and precise representation for further investigations. In this chapter, the formalization investigation on security design patterns using Codecharts will be elaborated and discussed. This includes highlighting the issues in the current security patterns catalogues and descriptions. Furthermore, an analysis of reasons, which make Codecharts might outperform other formal and informal representations of the structure of security design patterns, will be shown.

In addition, this chapter demonstrates the methodology followed in this research in order to formalise structures of security patterns from the original descriptions. This includes showing the linkage of the descriptions statement with the LePUS3 formulas and Codecharts notations. Finally, the outcomes of using this research methodology of formalising a number of security patterns from different security patterns catalogues will be shown.

4.1 Why security patterns as Codecharts

During the process of software development, many people are involved, such as software architects, security engineers, designers, implementers, maintainers and others. Each of those might not have enough knowledge about the others' views on points which they all share and work on. For example, the implementer and/or designer might understand that the security engineer needs to secure the communication with the client by using a password authentication pattern. However, the implementer and/or designer might have insufficient knowledge or have a different understanding about the password authentication pattern. This might introduce unnecessary confusion, which costs time an effort. Therefore, it is essential to clearly deliver the knowledge of the security engineer to the implementer and/or designer.

This would promote many aspects in all the software development stages. In the case of security, it will enhance the overall security of the system as it assures that all people in the development process share the same understanding about the specific security matter. In addition, it will allow the validation and/or verification team to check the implementation against clear points. It will also improve the process of maintaining the software as it will be easier to locate the aspects that the security engineer delivered to the designer, implementer, verification team and finally the maintainer [106].

In order to convey expertise, the patterns were introduced. Security design patterns are solutions to solve reoccurring security problems in a specific domain [9], [26]. In principle, security patterns are design patterns but they are for a specific domain, namely the security domain. Security patterns have gained a growing amount of interest. As a result, many researchers have introduced much research to describe, classify, detect, and verify security patterns.

In the case of security patterns descriptions, many have introduced patterns in many templates in order to improve the current descriptions of security patterns [1], [9], [26], [28], [29], [106]. The templates are similar to some extent. They usually combine the use of an informal description (expressed in the English language) with the use of UML class diagrams and sequence diagrams to demonstrate the structure and the behaviour, respectively, of a pattern. However, the descriptions of security patterns still suffer from a number of issues which can be noticed when putting them into practice. These issues are lack of precision, ambiguity and inconsistency between different descriptions of the same pattern.

Lack of precision and ambiguity are shown in all well-known security patterns catalogues in many forms. One example is the use of a natural human language to describe the pattern. It is obvious that the human language can be interpreted differently form one person to another. This makes it difficult to assure the identical understanding of the pattern by all the people involved.

In order to allow for understanding, implementing, verifying and/or detecting instances of security patterns, it is important to model them in a graphical, formal and elegant design description language. UML is a graphical modelling language. This makes it more useful and comprehensive when describing any design pattern. In other words, it is better than using a mathematical or textual representation for a pattern solution. However, UML diagrams are not formal, and therefore they cannot be used for rigorous reasoning for the patterns.

This research proposes the use of Codecharts for modelling the patterns, in particular security patterns. By modelling security patterns using Codecharts, the architectural and structural aspects of the patterns solution are to be modelled. A number of reasons are behind our choice of using Codecharts. These reasons are as follows:

1. Modelling abstraction;
2. Formality and tool support; and
3. Modelling Security properties.

4.1.1 Modelling abstraction

Abstraction is an important property in the modelling language. It is tangibly beneficial especially when modelling a large-scale software as well as when taking place in patterns. Codecharts has a number of elements (vocabulary) which demonstrate abstraction when modelling a pattern. For the context of modelling security patterns, the following elements are the interests and the focus (vocabulary): a) 1 -Dim Hierarchy; b) 1-Dim Signature; c) 1-Dim Class. While studying security patterns catalogues and descriptions, it was noticed that some patterns are described in an abstract way.

An example of abstraction is that in the description of the Intercepting Validator Pattern in the statement of “Intercepting Validator retrieves the appropriate validators according to the configuration for the target” [26]. This is clearly indicating the use of dynamic binding, especially as it is also stated, in the description, that “Intercepting Validator invokes a series of validators” and “Each Validator validates and scrubs the request data, if necessary". In the UML class diagram modelling this statement has gone far too abstract as shown in Figure 18. It can be seen that in order to model this statement in Figure 18, Intercepting Validator has a dependency relation with Validator labelled by “creates”. This might lead to the misunderstanding that there is only one class called “Validator” and it encapsulates all the validation process.

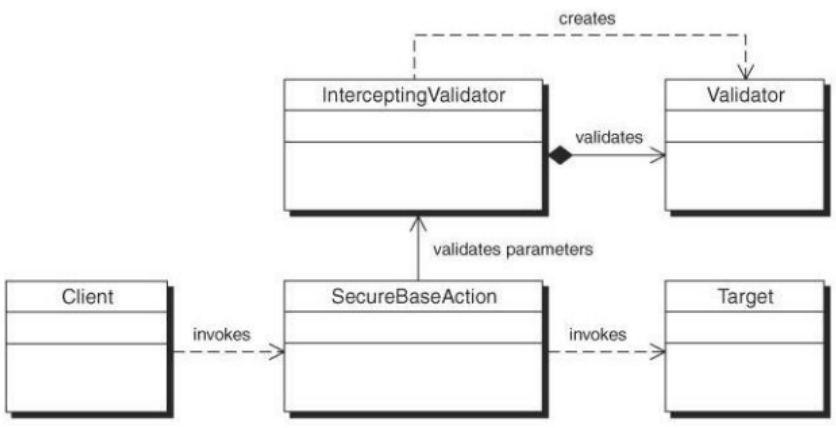


Figure 18: UML class diagram: Intercepting Validator Pattern [26]

If considering modelling them in a better way using UML, the above discussed statements in Intercepting Validator Pattern description could be modelled as shown in Figure 19. However, this would also not help with capturing the abstraction of the statement in the pattern's description. It is clear that 2 more class symbols (besides the existing one) are introduced to illustrate the idea of different validators. On the other hand, in Codecharts, it is possible to model the same statement using only one symbol which is a 1-Dim Hierarchy symbol as shown in Figure 20.

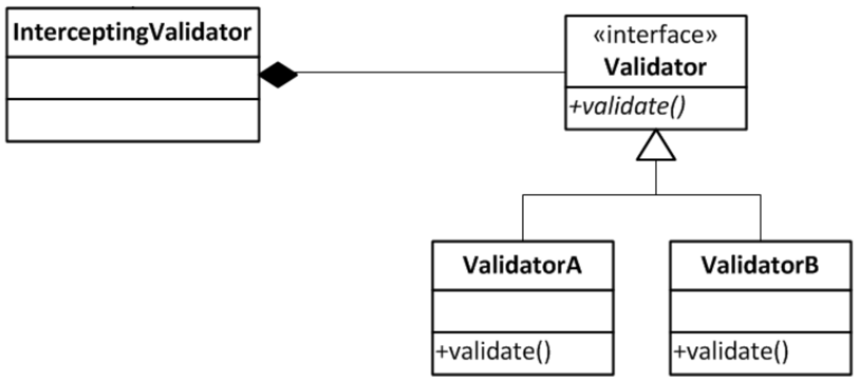


Figure 19: UML class diagram for modelling a different Validator in the Intercepting Validator pattern

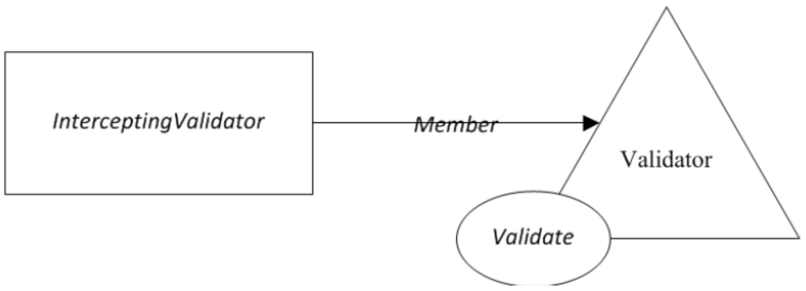


Figure 20: Codechart modelling a different Validator in the Intercepting Validator Pattern

In conclusion, if the UML class diagram does not reflect the patterns structure description, which is informal statements, this will introduce an issue of precision. The UML class diagram is a good way to represent the structure of a pattern. However, it does not help with modelling abstract structure effectively, whereas Codecharts can model the abstract structure in a more elegant and effective manner, as the latter provides few symbols which can be used to represent abstract aspects of the pattern's structure.

4.1.2 Formality and tool support

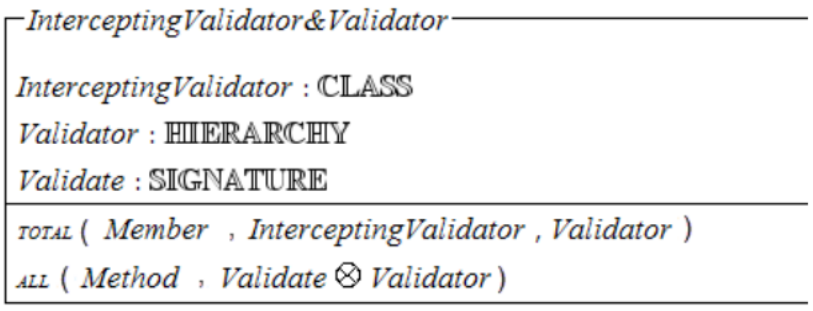
Formality is important when rigorous reasoning is needed. For example, in order to verify that a pattern is implemented in a source code, it is necessary to have the pattern formally represented. Security patterns are described using informal human language statements and UML diagrams. However, UML is an informal modelling language [56]. This means that is not based on formal semantics. Therefore, it cannot be used to represent the design or security design patterns formally.

Formal representation of the pattern allows many things to be carried out on it. For example, it allows searching for the instance of the pattern in the source code. It also allows for verifying source code conformance to the pattern as mentioned earlier. However, many researchers [55], [57], [59]–[62], [64], [80] have introduced their approaches to add formality to the UML diagrams. Several formal languages have been proposed to formalise the UML semantics. The most dominant languages are OCL, Description Logic (DL) languages and Z. However, these languages differ in their capability of representing UML properties.

For the context of formally representing the structure of the security pattern form a UML class diagram, using these languages introduces a risk of losing information as the languages might not be able to capture some UML properties, for instance, UML dependency in DLs languages [55] and interface notation in Z [86].

Another issue with using these languages is the problem of tools supporting automatic translation of the UML class diagram to the desired formal language. In the case of DL languages, the most well-known tools are FACT [83] and RACER [84]. However, these tools are suffering from a considerable efficiency issue as well as they do not support all UML properties [64]. In the case of using Z language, many have introduced their approaches to translate the UML class diagram to Z. However, these approaches have not been combined with tools to automate the translation process, or the tools are not available to be tried and tested.

On the other hand, Codecharts diagrams depend on (are based on) a formal language, which is First Order Predicate Logic (FOPL). Each element shown on the Codecharts diagram (Codechart) has a formal and semantic meaning. So, this shortcuts the process of detecting instances of the pattern structure as there will be no need for finding a suitable translation language and a tool for automating the translation process. Moreover, Codecharts has a tool support, the TTP Toolkit [13], which offers functionality such as automatic translation of the Codecharts diagram from graphical representation into LePUS3 and Class-Z formal schema, model extraction of java-based source code, and verification of Codecharts diagrams against source code. In order to show this tool in action, the formal representation of Figure 20 has been auto-generated and the result obtained from the tool is illustrated in Figure 21.


InterceptingValidator&Validator

InterceptingValidator : $\mathbb{CLASS}$
Validator : $\mathbb{HIERARCHY}$
Validate : $\mathbb{SIGNATURE}$

TOTAL (*Member* , *InterceptingValidator* , *Validator*)
ALL (*Method* , *Validate* $\otimes$ *Validator*)

Figure 21: Auto-generated LePUS3 and Class-Z schema

4.1.3 Modelling Security properties

The most important and key feature that Codecharts can offer for modelling patterns, especially security patterns, is the ability to model and identify the secured components in the patterns and/or the system. These secured components are usually identified as components within the system boundary. It can be seen in Figure 22 that the secured components are gathered and surrounded by the -rectangle which is labelled "System".

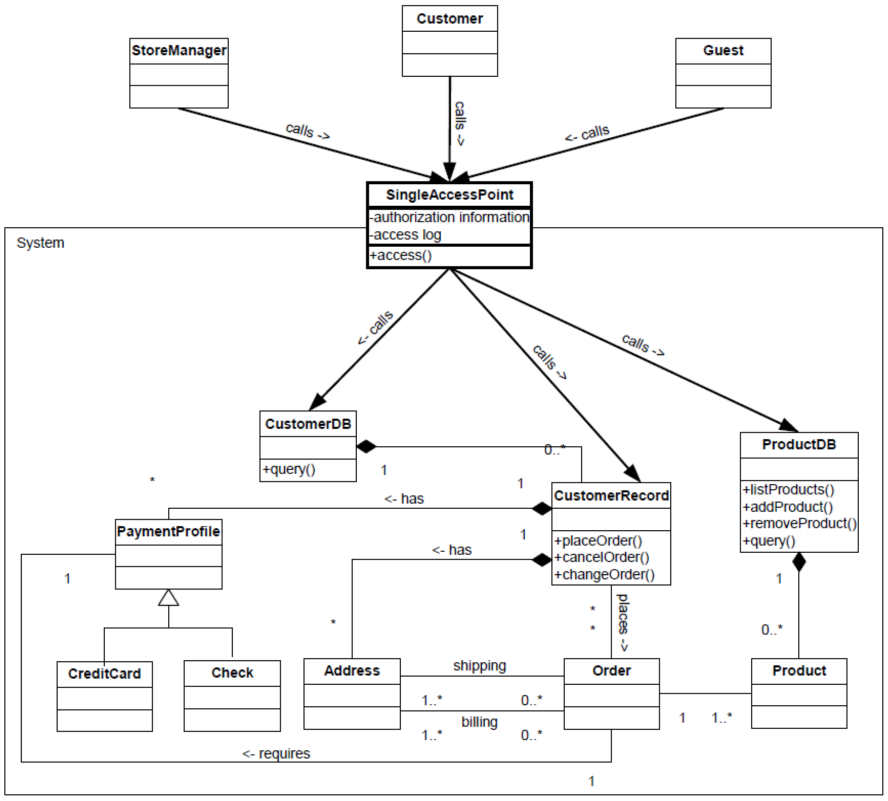


Figure 22: UML class diagram: Single Access Point [106]

Codecharts can be used to identify the secured components and formally model the system boundary which other languages cannot. Codecharts has an interesting symbol, called exclusivity symbol, which indicates either LEFT exclusive or RIGHT exclusive predicate [38]. This symbol serves information hiding or neglecting when modelling a pattern. For example, in the description of Single Access Point Pattern (SAP) in [106], it is expressed that the internal components can communicate with each other and/or with the single access point; however, external entities cannot communicate with the internal components except via a single access point participant in Figure 22. This has not been represented effectively in the UML diagram. In addition, it might not be possible to translate into any formal language as the UML class diagram has no notation expressing such a constraint. This issue can be solved when using Codecharts to model the pattern as shown in Figure 24.

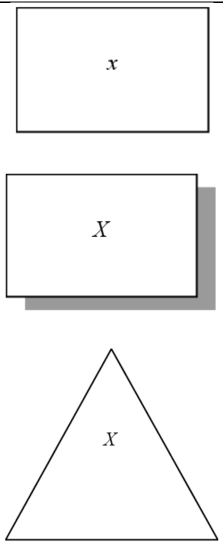
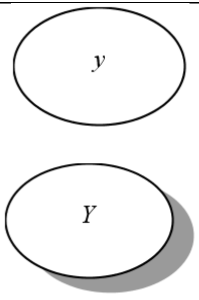
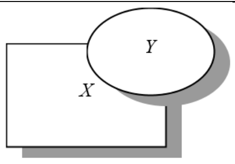
4.2 Our Formalisation methodology

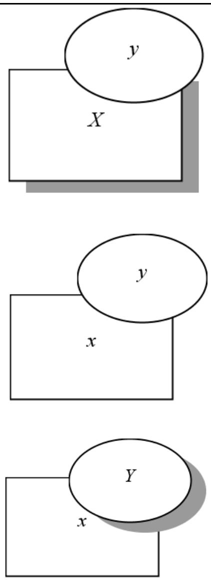
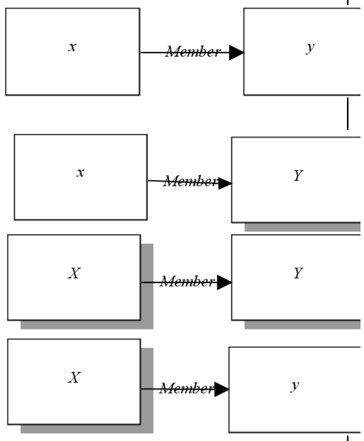
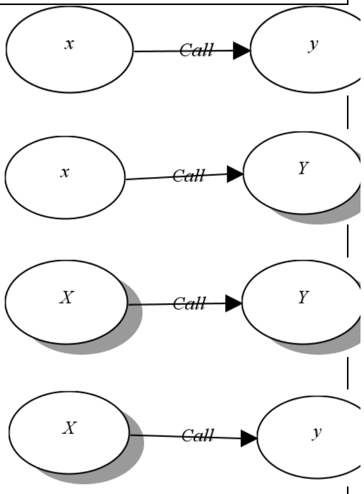
In this research, an approach have developed in order to be used when formalising any security patterns. This approach was developed after thorough analysis of a number of well-known security patterns catalogues [1], [9], [26], [28], [106]. The main outcome of this analysis was this approach, which aims to link the informal statements in the patterns description with the formal Codecharts notations.

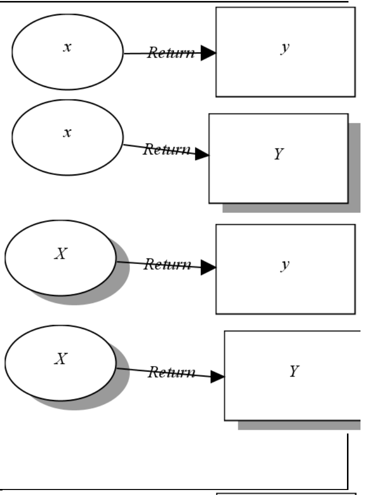
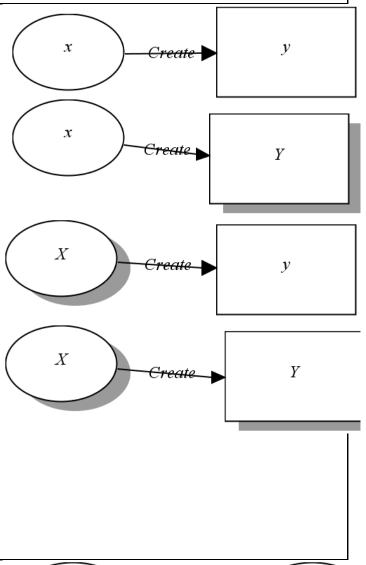
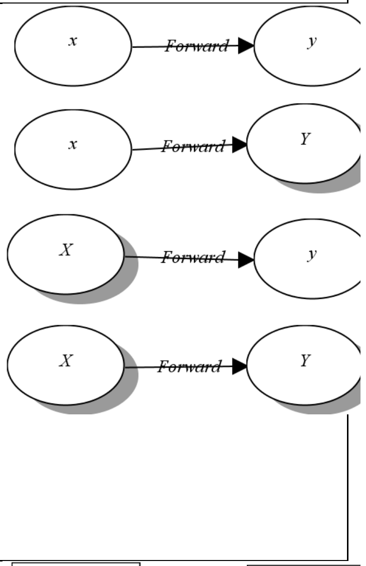
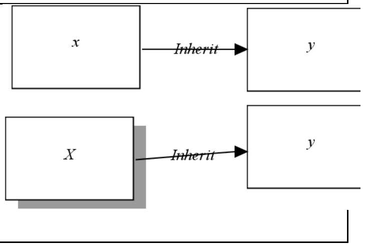
The approach uses a pattern description which should be following the security patterns description template used currently in the existing catalogues [1], [9]. The approach is meant to take the "*Solution*" section of the pattern description and to analyse it. Then, it takes the clear and precise statements and converts them to Codecharts notations.

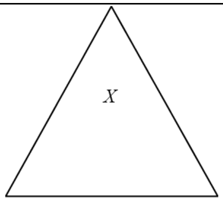
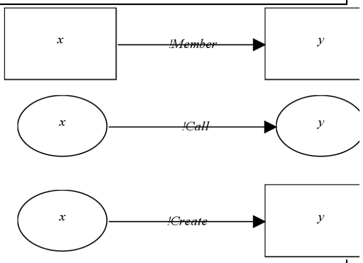
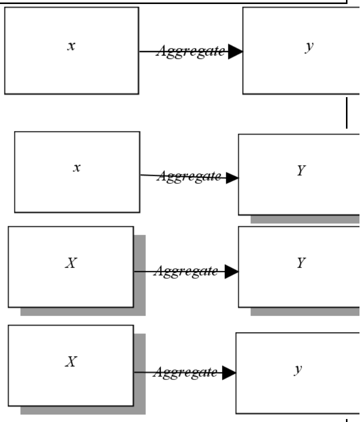
The precise statements are identified by the clear key words shown in the Table 3. These key words are the guidelines for our approach in the process of formalising the statements in the descriptions of patterns into Codecharts notations. Consequently, the notations form a single Codecharts.

Table 3: Our methodology for formalising security patterns from descriptions of pattern structure

Informal Statements	LePUS3 formula	Codechart notation (visual notations)
: - Participant: - x:…… - … ***consists of … classes***…	$Class(x)$ or $ALL(Class, X)$	
- x ***implements*** a method to … - x ***provides*** an interface … - x ***defines a method..*** - x ***provides*** a set of actions	$Signature(y)$ or $ALL(Signature, Y)$	
- x ***implements*** a method to … - x ***provides*** an interface … - x ***defines a method..*** - …..x ***provides*** a set of actions -	$x \otimes y$ Or $x \otimes Y$ Or $X \otimes y$	

	Or $X \otimes Y$	
- x ***uses*** y - x ***defines*** an instance of y - x ***keep/hold a reference*** of y -	$Member(x, y)$ Or $TOTAL\ (Member, x, Y)$ Or $TOTAL\ (Member, X, Y)$ Or $TOTAL\ (Member, X, y)$	 x Member y x Member Y X Member Y X Member y
- x ***invokes*** y's method - x ***triggers*** actions… - x ***submits*** the request -	$Call(x, y)$ Or $TOTAL\ (Call, x, Y)$ Or $TOTAL\ (Call, X, Y)$ Or $TOTAL\ (Call, X, y)$	 x Call y x Call Y X Call Y X Call y

- x ***returns*** a generic failed login message. - x ***provides*** information about its users… - x ***grants*** messages access	$Return(x, y)$ Or $TOTAL\ (Return, x, Y)$ Or $TOTAL\ (Return, X, y)$ Or $TOTAL\ (Return, X, Y)$	
- x ***defines*** an instance of y - x ***creates*** a y instance -	$Create(x, y)$ Or $TOTAL\ (Create, x, Y)$ Or $TOTAL\ (Create, X, y)$ Or $TOTAL\ (Create, X, Y)$	
- x ***passes*** the y the request….. - x ***forwards*** data to…	$Forward(x, y)$ Or $TOTAL\ (Forward, x, Y)$ Or $TOTAL\ (Forward, X, y)$ Or $TOTAL\ (Forward, X, Y)$	
- x ***provides*** a means for ***separating responsibility*** through ***inheritance***	$Inherit(x, y)$ Or $TOTAL\ (Inherit, X, y)$	

- x ***inherits*** from y - x can be ***nested*** so that more senior roles ***inherit*** the privileges of junior roles. - ***Define*** or re-use an ***interface*** to be used by your CHECKPOINT …… ***Implement*** required ***concrete*** CHECKPOINT …… ***at least one concrete*** CHECKPOINT ***implementation*** is needed	Or $ALL(Class, X)$	
- x ***prevents external*** entities from ***communicating*** directly with y in the system - ***communication*** between x and the y may ***not be overheard*** by external entities. - x ***defines one single*** interface for all ***communication*** with system external entities -	$LEFTEXCLUSIVE(Member, x$ Or $LEFTEXCLUSIVE(Call, x, y)$ Or $LEFTEXCLUSIVE(Create, x, y$	
- x ***contains a set of*** privileges -	$Aggregate(x, y)$ Or $TOTAL\ (Aggregate, x, Y)$ Or $TOTAL\ (Aggregate, X, Y)$ Or $TOTAL\ (Aggregate, X, y)$	 x Aggregate y x Aggregate Y X Aggregate Y X Aggregate y

As it can be seen in Table 3, the key words are taken from the patterns description statements, which are commonly used. The second and the third columns of the above table show the formal interpretations of the statements having the key words in the form of LePU3 formulas and their graphical notations in Codecharts.

Finally, in order to formalise the security patterns, the above approach has been adopted in this research. The following section shows and discusses a number of formalised security patterns. The diagrams in the following section are the outcomes of the approach, shown in this section, and the pattern descriptions that this research investigated.

4.3 Formalised security patterns

Using our methodology shown in the previous section, a number of security design patterns from different security patterns catalogues have been formalised using Codecharts. This section shows and discusses the formalised patterns as Codecharts, which are the outcomes of using our methodology to analyse the patterns descriptions and formulise the structure and architecture of the investigated patterns. The number of the formalised patterns using Codecharts is more than 16 patterns from 5 different catalogues [9], [26], [28], [106], [107].

In this section, some of the patterns which have been formalised using Codecharts are selected in order to discuss the features and the benefits of using Codecharts to formally model security patterns. Namely, Single Access Point pattern (Figure 24) and Intercepting Validator pattern (Figure 26) are selected to be shown and discussed. More formally modelled security patterns can be found in APPENDIX A.

4.3.1 Single Access Point pattern in Codecharts

Single Access Point (SAP) is one of the well-known security design patterns. It is implemented in many secured systems. Wassermann et al. [106] has described SAP using the common design patterns description catalogue. Table 4 abbreviated the pattern description from its original catalogue. In addition, Figure 23 illustrates the UML class diagram which has been used in the original catalogue to demonstrate the patterns structure and architecture.

Table 4: Single Access Point pattern [106] (abbreviated)

Intent	Single Access Point pattern limits all communications form outside to one single interface in order to control and monitor.
Structure	The Single Access Point represents the systems' only connection to the outside. All incoming communication requests are passed to the Single Access Point instance. From there, they will be directed to the intended recipient, if all security relevant requirements are met. …… The class Single Access Point (SAP) is the only one that interacts with external entities.
Participants	External Entities: they are components located outside the systems boundary. They contact the SAP in order to communicate with internal entities. Internal Entities: they are all components located inside the systems boundary. Single Access Point: it provides an interface that allows external parties to communicate with system internal components, gathers information about the occurring access requests, their origin and authorisation information, triggers actions or forwards data to parties inside the system.

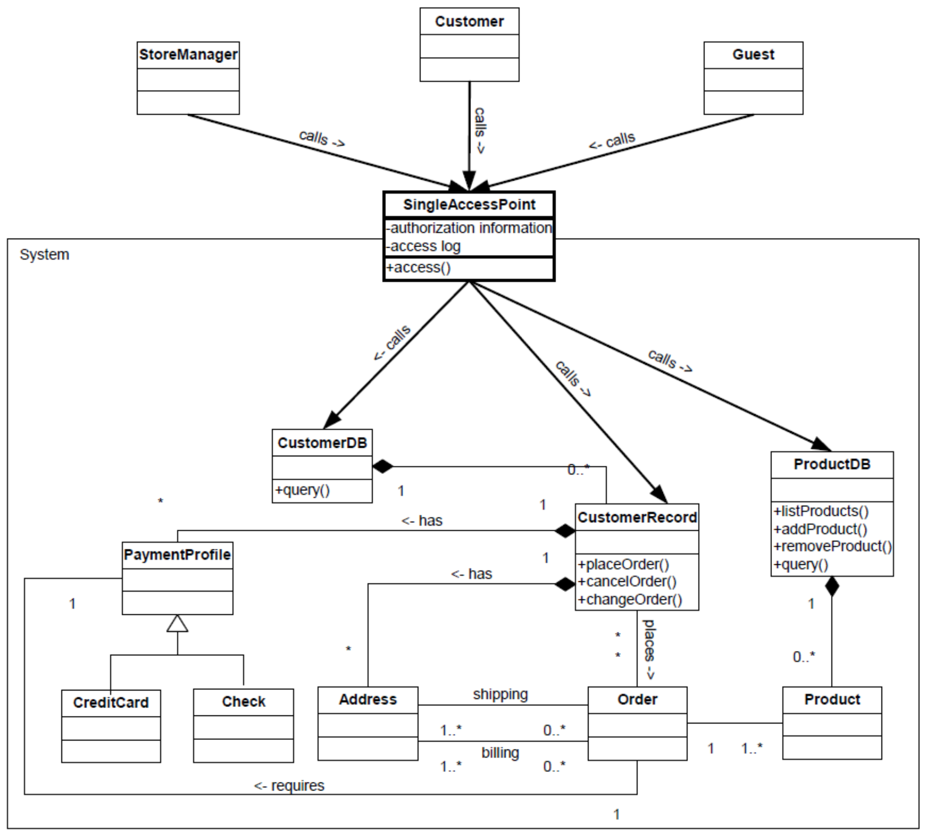


Figure 23:UML class diagram: Single Access Point Pattern [106]

As it can be seen from the diagram in Figure 23, not all the statements of the pattern structure and architecture are depicted in the diagram. For instance, the statement "*The class Single Access Point (SAP) is the only one that interacts with external entities*" demonstrates a security property which can be captured using Codecharts by means of exclusive operator (Exclamation symbol) as can be seen in Figure 24.

Exclusive operator (Exclamation symbol) appears in Figure 24 on the Call relation symbol between the elements named Access and secureAction. It represents an important feature which can be used to present "Access Control" in security patterns.

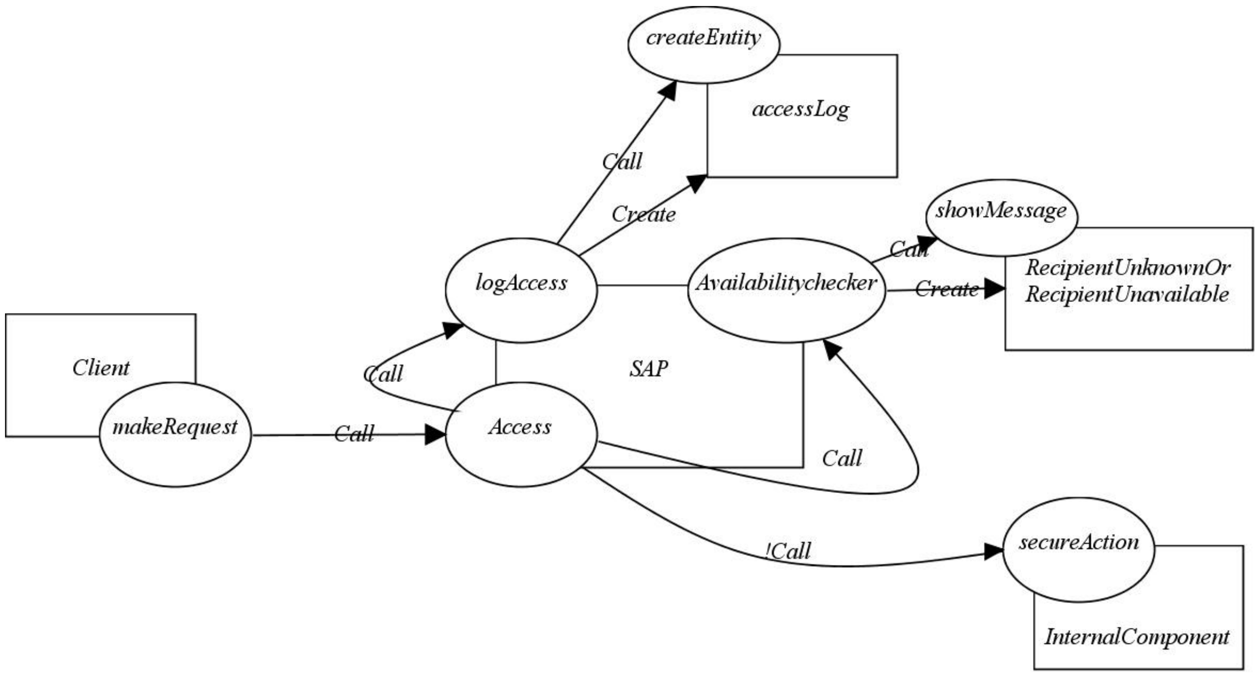


Figure 24: Codecharts: Single Access Point pattern (SAP)

Table 5 gives a brief description on the elements shown in Figure 24. This will clarify more on how Codecharts outperform UML the class diagram in depicting patterns architecture. Furthermore, abstraction in the description of the patterns structure is detailed in a misleading manner. For example, the “Internal Entities” are illustrated in Figure 23 by three named classes and operations in classes (CustomerDB, CustomerRecord, and ProductDB). This will be additionally solved when using Codecharts by means of class variables and signature variables as can be seen in Figure 24.

Table 5: Explanation of tokens in Figure 24

Class variables (Client, SAP, accessLog, InternalComponent, RecipientUnknownOrRecipient Unavailable)	Represent classes and interfaces.
Binary relations (Call)	Represent relation between components.
Signatures variables (Access, SecureAction, logAccess, AvailabilityChecker, showMessage, createEntity)	represent signatures of methods in classes
A signature over a class (superimposition)	is called a superimposition which represents a method. An example of such superimposition is Access superimposed on SAP which is a superimposition representing a method whose signature is Access and is in class SAP.
! symbol	shown on the Call relation between Access (superimposed on SAP) and SecureAction (superimposed on InternalComponent) represent Left Exclusive operator which means if and only if SecureAction (superimposed on InternalComponent) is in Call relation and SecureAction (superimposed on InternalComponent) is the callee then the caller must be Access (superimposed on SAP).

4.3.2 Intercepting Validator pattern in Codecharts

Intercepting Validator is also a well-known security design pattern. It has been introduced to solve the problems of requests that contain invalid or harmful data. One example of Intercepting validator application is the validator of SQL injections. So, the main aim of this pattern is to check the data before processing it in order to make sure that no attack codes are included in the requests.

Alur et al. [26] have described Intercepting validator pattern in their catalogue. Although the authors of this catalogue aimed to be as clear as possible with reserving the abstraction level of patterns, some ambiguity can be noticed in the catalogue. In addition, ambiguity can be noticed in the UML diagrams, which are used to describe patterns aspects.

A UML class diagram is used to present the pattern structure and architecture. This diagram is shown in Figure 25. However, with a glance at Figure 25, it can be seen clearly that the diagram suffers from ambiguity. For instance, it can be seen that the SecureBaseAction class has a relation labelled “validates parameters” with the InterceptingValidator class. This relation can be interpreted differently by implementer. In addition, the diagrams only show classes (boxes) with nothing inside. Therefore, these are an ambiguous way to present the pattern.

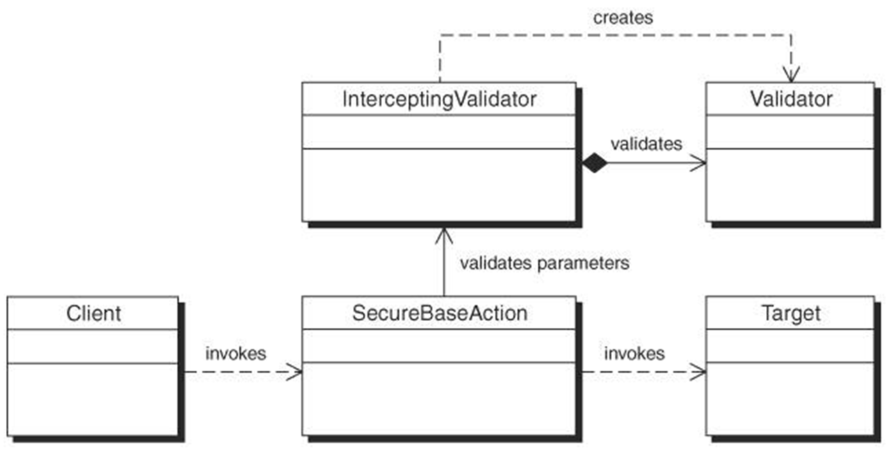


Figure 25: UML class diagram: Intercepting Validator Pattern [26]

Furthermore, it is important to ask whether the diagram reflects the description itself. The following quote is from the same pattern description which is used Figure 25 to present the pattern structure. However, the level of object-oriented abstraction shown in the statements of the quote could not be presented. This is having an "abstract" Validator class to represent all types of validators. Even if this was shown, it would be shown using at least three tokens (validators), as can be seen in Figure 19.

"*InterceptingValidator retrieves the appropriate validators according to the configuration for the target. InterceptingValidator invokes a series of validators as configured. Each Validator validates and scrubs the request data, if necessary.*" [26]

The aforementioned issues can be solved when using Codecharts. Therefore, the Intercepting Validator pattern description [26] has been analysed and the formal modelling of this pattern is introduced and demonstrated in Figure 26. Unlike Figure 25, each element shown in Figure 26 does have only one meaning and a formal representation. This improves the precision; however, it does not impact the level of abstraction shown in the pattern description.

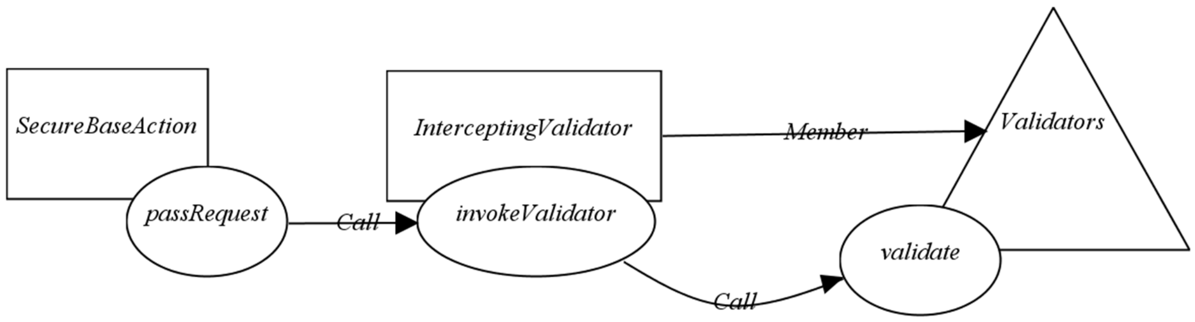


Figure 26: Codecharts: Intercepting Validator pattern

As can be seen from Figure 26, the issues of modelling the abstraction in the pattern description formally, precisely, and concisely can be solved using the Hierarchy variable, which represents any set of classes that has a superclass. The other notations in Figure 26 are similar to the ones that have been explained earlier when Figure 24 was discussed. However, for more details and explanations on the notations of Codecharts, these [38], [108] would provide a useful manual and references on Codecharts and LePUS3.

4.3.3 Formality in the Codecharts of patterns

Although Single Access Point and Intercepting Validator patterns shown in Figure 24 and Figure 26 are visually presented in Codecharts, this does not mean that these two types of modelling are not formal modelling. Figure 27 illustrates the formality behind the visual modelling of SAP pattern shown in Figure 24. In addition, Figure 28 demonstrates the formalisation of the visual modelling of Intercepting Validator pattern shown in Figure 26.

```
SAP
Client , InternalComponent , RecipientUnavailableOrRecipientUnknown , SAP , accessLog : CLASS
Access , Availabilitychecker , createEntity , logAccess , makeRequest : SIGNATURE
secureAction , showMessage : SIGNATURE
----------------------------------------------------------------
Call ( Access ⊗ SAP , logAccess ⊗ SAP )
LEFTEXCLUSIVE ( Call , Access ⊗ SAP , secureAction ⊗ InternalComponent )
Call ( Access ⊗ SAP , Availabilitychecker ⊗ SAP )
Call ( Availabilitychecker ⊗ SAP , showMessage ⊗ RecipientUnavailableOrRecipientUnknown )
Call ( logAccess ⊗ SAP , createEntity ⊗ accessLog )
Call ( makeRequest ⊗ Client , Access ⊗ SAP )
Create ( Availabilitychecker ⊗ SAP , RecipientUnavailableOrRecipientUnknown )
Create ( logAccess ⊗ SAP , accessLog )
```

Figure 27: LePUS3 and Class-Z schema of SAP

```
Intercepting Validator Pattern
InterceptingValidator , SecureBaseAction : CLASS
Validators : HIERARCHY
invokeValidator , passRequest , validate : SIGNATURE
----------------------------------------------------------------
TOTAL ( Call , invokeValidator ⊗ InterceptingValidator , validate ⊗ Validators )
Call ( passRequest ⊗ SecureBaseAction , invokeValidator ⊗ InterceptingValidator )
TOTAL ( Member , InterceptingValidator , Validators )
```

Figure 28: LePUS3 and Class-Z schema of Intercepting Validator

It is important to mention here that these formal schemas are not the outputs of Codecharts transformation. In fact, these are the formal face of the shown Codecharts. Furthermore, in order to auto-generate these schemas from the Codecharts, an advantage was taken of the TTP Toolkit [13], which is the tool support of Codecharts.

4.4 Summary

In this chapter, issues in the current security patterns catalogues have been discussed. These are informality, lack of precision, and ambiguity. Furthermore, critical analysis has been shown in order to highlight and explain the reasons which might make Codecharts outperform other formal and informal representations of the structure of security design patterns.

Furthermore, in this chapter, the methodology of formalising structures of security patterns from the original descriptions has been demonstrated. This includes showing the linkage of the descriptions statement with the LePUS3 formulas and Codecharts notations. Finally, the outcomes of using our methodology for formalising a number of security design patterns from different security patterns catalogues have been shown.

The number of security design patterns formalised using Codecharts are more than 16 patterns. Single Access Point and Intercepting Validator have been shown and discussed in this chapter. However, more formally modelled security patterns can be found in APPENDIX A. The Codecharts of those patterns will be used as inputs for further investigation of the security patterns. The investigation will be carried out on analysing the relations among patterns, patterns variations, conformance checking, and pattern detection.

5 Variations in Security patterns and relations among patterns

Note: some parts of this chapter were published in:

Alzahrani, A.A.H., A.H. Eden, and M.Z. Yafi. 2014. 'Structural Analysis of the Check Point Pattern'. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*, 404–8. doi:10.1109/SOSE.2014.56.

Many issues exist in the current security pattern catalogues. For example, some of the current catalogues [9], [27], [109] describe the same patterns such as Single Access Point, Check Point, Full View With Error, and others; however, are the descriptions of the same patterns consistent? What are the consequences of not being consistent?

Another issue, which exists in the current security catalogues, is the claims of relations among security patterns and with design patterns. These claims are often shown in a designated section, which is the patterns description template called "Related Pattern". Indeed, it is useful to know what other patterns are related to the pattern at hand when studying a pattern. However, having these "strong" claims of relations without validating and investigating them might result in incorrect implementation of the patterns.

Therefore, this chapter discusses the issues of the inconsistency of descriptions of the same patterns in security pattern catalogues. In addition, the problem, of the non-validated claims of relations among security patterns and with design patterns, will be highlighted and discussed. Our investigation of the above mentioned issues is conducted using Codecharts and a number of security patterns.

5.1 Variations in Security patterns

Security patterns are to be implemented at the end. Therefore, they have been described and shown in catalogues to allow any interested person to understand them and apply them in systems which need to overcome a security issue that patterns are authored for. A number of security patterns catalogues are describing the same patterns. For instance, Single Access Point is described in [9], [27], [109]. However, are the descriptions consistent?

Security patterns catalogues describe patterns differently. For example, one catalogue emphasises more in the description of Single Access Point on the structure [27]. Another emphasises more in the description of the pattern on the concrete application and dynamic behaviour [9]. Others emphasise more on showing the pattern as a part of a framework which combines a set of security patterns [109].

Although each catalogue describes the pattern in a common template, the catalogues fail to be consistent in describing the same pattern. Inconsistency raises many issues such the issue of sharing the same understanding among the people involved in the system development life cycle. In order to ensure the correct implementation of a pattern, it is important that implementers, designers, security architects, verification and validation teams, and maintainers share the same understanding of the pattern.

Furthermore, precision is important in reasoning out the security pattern. Here, the term ‘reasoning’ means the verification of the implementation of the pattern in the source code. Moreover, when automating this verification process, the need for precision emerges again. Therefore, the current issue of inconsistency in describing the same patterns needs to be tackled.

In order to tackle the above mentioned issues, the patterns need to be formalised with one of the formalisation languages. When aiming to formalise a pattern from one of the catalogues, it is important to know what to formalise. Obviously, security patterns catalogues share a similar description template [9], [26]–[28], [107], [109]. This template has two important sections which are the problem (pattern motivation) and the solution (pattern structure and behaviours).

Formalisation of a security pattern focuses on the solution section for pattern description. This section gathers statements and often the UML diagram describing the patterns. The information in this section is used to understand and implement the pattern. Therefore, formalising the information in this section reflects formalising the pattern.

In this research direction, Codecharts have been chosen for formalising the patterns. The choice of Codechart has been discussed and explained in former chapter in the section entitled Security Patterns Formalisation. However, it is important to mention here that the formalisation will be carried out on the static structure described in the solution section of the pattern description. A number of reasons are behind this. First, Codecharts are limited to formalise static structure. Second, formalising the structure of the pattern provides the base for behaviour formalisation. Third, the information about structure is richer than the information about behaviour as the former gathers much core information such as inheritance, dependency, production, etc.

Back to security patterns catalogues, it can be easily noticed that there are differences in the descriptions of the same pattern. Having Check Point Pattern as a case study, the pattern is described in [9] and [27] differently. Using Codecharts to formalise and analyse the pattern, it can be proven formally and visually that the pattern is not consistent in the two catalogues. This has led us to the term "*pattern variant*". A pattern variant can be regarded as a different version of a pattern which has been standardised in a number of security patterns catalogues.

Bayley et al. [110] have addressed this issue of specifying patterns variants in their work of formalising design patterns using FOL and GEBNF. The authors have introduced an approach in order to improve on their previous work [111] of formalising structure of patterns. This approach was to improve on the accuracy on patters specifications as well as to tackle ambiguity in informal descriptions of patterns. Furthermore, it is used to reports on patterns variations.

Bayley et al. introduced a use of a set of keywords to allow specifying a pattern and its variants. They keywords are: "*Optional*", "*Alternatives*", "*Depends on*", and "*In case of*". Each of which has a definition in their approach. Despite the fact that this approach offers a clearer way of specifying a pattern with all variants in ONE formalization, the specification of a pattern with variants is still textual. Moreover, without a good background in logic notations, these specifications cannot be unpacked. On the other hand, the research in this thesis aims to specify variants of security patterns using a visual, however, formal specification language (Codecharts). Visual notation allows easier understanding of patterns and the differences among variants. Furthermore, as these visual notations (Codecharts) are based on a formal foundation of FOL predicates, variants can be formally presented, analysed, and evolved.

In addition, in the case of carrying rigorous reasoning (such as pattern detection), Bayley et al. formalization of any pattern needs to be unpack and variants needs to be separated as reasoning would be conducted for each variant. Therefore, as the aim in this thesis is to use the formalization of patterns in order to detect patterns in the source code, the formalization in this thesis shows pattern variants separately.

As a case study aimed to show the pattern variants, the Check Point Pattern, which is described in [9], has been studied in order to show the main difference in the structure of the pattern with the description in [27], Figure 29 and the following statement are taken from the description of the pattern.

“*This CHECK POINT interface corresponds to the abstract strategy in STRATEGY [GoF95]. The interface will provide hooks for I&A, authorization, handling unsuccessful attempts.*” [9].

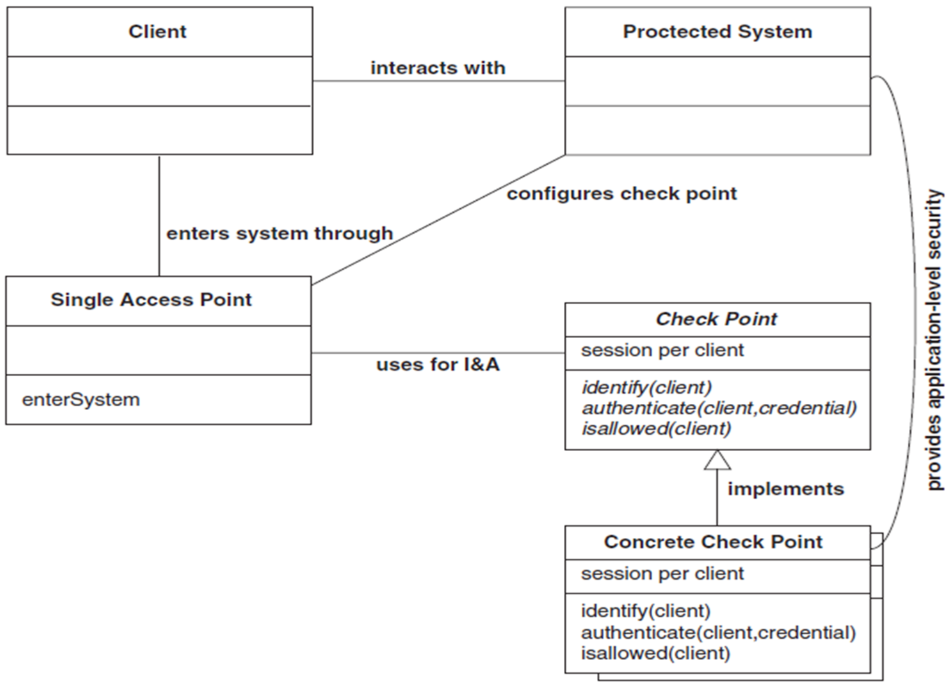


Figure 29: Check Point (with hierarchy) structure [9]

The description has been analysed and formalised using Codecharts. The results of the analysis and the formalisation of the pattern are shown in Figure 30. When Looking at the Codecharts shown in Figure 30 (which formalises the pattern described in [9]) and the Codecharts shown in Figure 31 (which formalises the pattern described in [27]), the difference can be clearly and visually seen between the pattern's variants. Check Point pattern has another variant which is with a hierarchy of "Concrete Check Point" as described in [9].

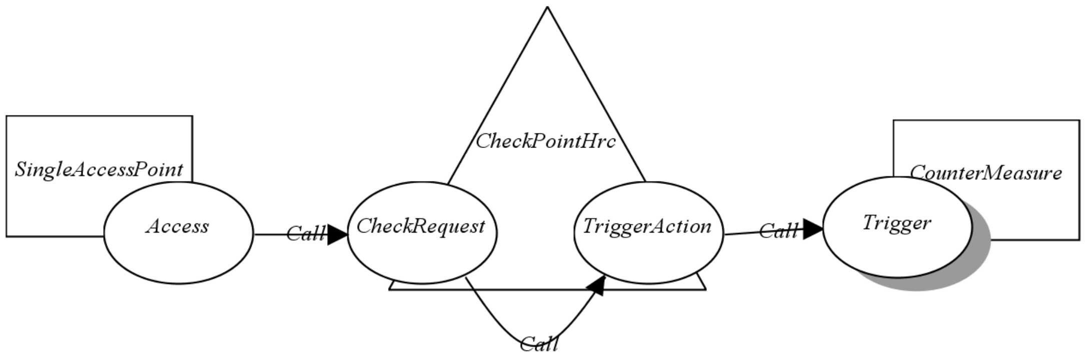


Figure 30: Codechart: Check Point pattern with hierarchy

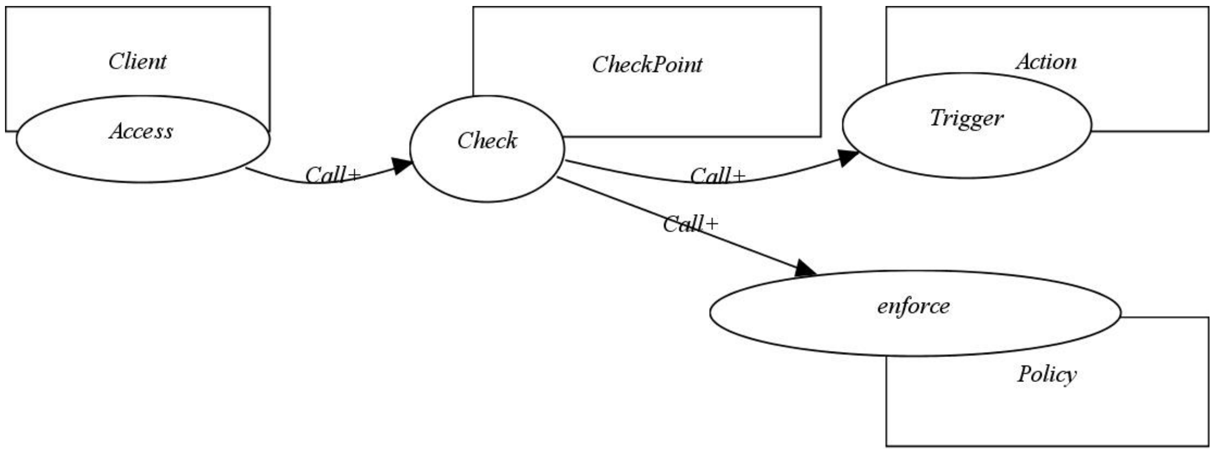


Figure 31: Codecharts: Check Point pattern (CP)

The Check Point patterns in the two catalogue in [9] and [27] are the same in general, however, they differ in a number of aspects when modelling them formally using Codecharts on the two descriptions. Starting with the similarity between them, first, the Check Point patterns in the two catalogues describe a participant called "*CheckPoint*" which implements a method for checking the received requests to authenticate and/or authorize the sender of the request. In addition, "*CheckPoint*" employs a "*Policy*" to carry out the authentication and authorization. Furthermore, they describe a "*CounterMeasure*" action that is triggered based on the results of the checking.

Moving to the differences between the Check Point patterns in the two catalogues, first, Schumacher et al. [9] Check Point variant in (Figure 30) demonstrates the encapsulation of "*Policy*" in the "*CheckPoinHrc*" participant as it does not regard it as a separate class, whereas, Wassermann et al. [27] pattern (Figure 31) separates it from the "*CheckPoint*" participant (with the class variable called "*Policy*"). Second, Schumacher et al. [9] Check Point variant in (Figure 30) defines an interface (Hierarchy notation) for "*CheckPoinHrc*" participant to allow implementing any number of "*CheckPoint*" each with a special policy entity, whereas, Wassermann et al. [27] pattern (Figure 31) does not allow such generality. It is important to mention that the "*Call+*" in Wassermann et al. [27] pattern (Figure 31) is a mean of generality in the pattern as the "+" denotes that the relations can be either direct or indirect, whereas, the "*Call*" and Schumacher et al. [9] Check Point variant in (Figure 30) restrict the relations to be only direct "*Call*" relations.

The method named "*CheckRequest*", which is responsible for checking the incoming request of access in Figure 30, does not directly trigger the countermeasure action; instead, it delegates this to another method in the concrete object of "*CheckPointHrc*" named "*TriggerAction*". Whereas, the Check Point pattern in Figure 31 includes the responsibility of triggering the countermeasure actions in the method of checking which is named "*Check*" in the class "*CheckPoint*".

In the Check Point variants shown in Figure 30 and Figure 31, the countermeasure are modelled as a class despite the naming used as these as just variables names base on the Codecharts notations (white box is a viable class). However, the difference between variants in the countermeasure is that the variant in Figure 30 specify two different points that variant in Figure 31 does not do. The first point is that "*CounterMeasure*" class has a set of methods ("Trigger") that each method in this set represents an action which can be called by "*TriggerAction*", whereas, the pattern variant shown in Figure 31 specify that the countermeasure ("*Action*") class has only one method ("*Trigger*") which represents the action to be triggered The second point is that "*TriggerAction*" can call any method in the set of methods "Trigger" which are defined in the class "*CounterMeasure*".

Moreover, from the formal specifications shown in Figure 32 and Figure 33, which translate the two Codecharts (Figure 30 and Figure 31) into LePUS3 and Class-Z formulas and specifications, the differences can be formally proven and shown. Consequently, it can be proven that the catalogues are not consistent and the two descriptions [9], [27] of the same pattern (Check Point pattern) are offering two variants of the pattern.

Check Point Pattern Wassermann et al. variant

$$
\begin{array}{l}
\mathit{Action}\ ,\ \mathit{CheckPoint}\ ,\ \mathit{Client}\ ,\ \mathit{Policy} : \mathbb{CLASS} \\
\mathit{Access}\ ,\ \mathit{Check}\ ,\ \mathit{Enforce}\ ,\ \mathit{Trigger} : \mathbb{SIGNATURE} \\
\hline
\mathit{Call}(\mathit{Access} \otimes \mathit{Client}\ ,\ \mathit{Check} \otimes \mathit{CheckPoint}) \\
\mathit{Call}(\mathit{Check} \otimes \mathit{CheckPoint}\ ,\ \mathit{Trigger} \otimes \mathit{Action}) \\
\mathit{Call}(\mathit{Check} \otimes \mathit{CheckPoint}\ ,\ \mathit{Enforce} \otimes \mathit{Policy})
\end{array}
$$

Figure 32: LePUS3 and Class-Z schema of Check Point Codechart shown in Figure 31

$$\begin{array}{|l}
\textit{Check Point Pattern Schumacher et al. variant} \\
\\
\textit{CounterMeasure}, \textit{SingleAccessPoint} : \mathbb{CLASS} \\
\textit{CheckPointHrc} : \mathbb{HIERARCHY} \\
\textit{Access}, \textit{CheckRequest}, \textit{TriggerAction} : \mathbb{SIGNATURE} \\
\textit{Trigger} : \mathcal{P}\,\mathbb{SIGNATURE} \\
\hline
\textsc{Total}(\textit{Call}, \textit{Access} \otimes \textit{SingleAccessPoint}, \textit{CheckRequest} \otimes \textit{CheckPointHrc}) \\
\textsc{Total}(\textit{Call}, \textit{CheckRequest} \otimes \textit{CheckPointHrc}, \textit{TriggerAction} \otimes \textit{CheckPointHrc}) \\
\textsc{Total}(\textit{Call}, \textit{TriggerAction} \otimes \textit{CheckPointHrc}, \textit{Trigger} \otimes \textit{CounterMeasure}) \\
\hline
\end{array}$$

Figure 33: LePUS3 and Class-Z schema of Check Point Codechart shown in Figure 30

With the above, it can be shown that using Codecharts can help depict the variants of the same pattern. This would assure that the same understanding of the pattern would be shared among the people involved in the development life cycle of any system in whose source code security patterns will be implemented, verified, maintained, and/or documented in its source code. Codecharts can be used as a formal input for an automating tool to reason out the patterns in a source code. One instance of this automated reasoning is the detection of the patterns from the source code.

5.2 Relations among security patterns and other design patterns

Relations among security patterns and with design patterns are often described in security patterns catalogues. In the catalogues [9], [26]–[28], [107], [109], the common pattern description template refers to the relation of the pattern at hand with other patterns in the designated subsection called “Related Patterns”. It is important to validate and explain these claimed relations among patterns in order to select and implement the patterns [2], [33], [112]–[114].

Many researchers [2], [33], [112]–[114] have proposed their approaches to find these relations either manually or automatically by investigating and studying the similarity among security and design patterns descriptions. However, validations of the actual relations are not covered sufficiently. Furthermore, the work done up-to-date is introducing more claims about relations among patterns.

Most of the studies [2], [33], [112]–[114] on relations among patterns are to check the similarity of patterns to one another. The similarity is calculated and claimed based on the comparison between the patterns descriptions sections such as Problem, Context, Forces, and/or Related Patterns sections.

Bayley et al. [115] have introduced their approach for formalising the relation of composition between design patterns. In order to do so, the authors have employs GEBNF in order to specify patterns separately, then introduced a set of operators to formalise the different kinds of compositions. The operators, which the authors have defined, are: *Restriction operator, Superposition operator, Extension operator, Flatten operator, Generalisation operator, Lift operator*. As a case study to demonstrate the approach, the authors have formalise the composition relation between *Builder* and *Composite* patterns as described in [18].

Similarly, this thesis aims to investigate the relations among patterns, however, there are a number of differences between Bayley et al. approach and this thesis approach. First, Bayley et al. approach aims to formalise the composition relations among patterns, whereas, this thesis aims to formalise and investigate all claims for relations generalisation, concrete cases, implementation, and composition relations. Current case studies of Bayley et al. approach are considering the relations among the design patterns only and in particular design patterns of [18] book, whereas, this thesis case studies focus on security patterns and their relations with design patterns ([18] book) and other security patterns from different sources.

Although Bayley et al. approach formalizations of composition are formal, they are still textual. On the other hand, this thesis employs the use of visual formalisation (Codecharts) in order to formalise the patterns separately, then formalise the composition relations visually and formally by the means of formal schemas as can be seen in Figure 35 and Figure 36.

In addition, in this thesis, validating the results of formalisation of the case studies is considered and done by the detection approach and reasoning on found instances of composition formalisation by use of TTP Toolkit verifier, whereas, this is not considered in Bayley et al. approach.

The used method to study and validate the claims of relations among patterns can be summarised as follows: first, claims of relations among patterns are gathered from patterns catalogues; second, patterns are formalised using Codecharts or imported if already formalised patterns in Codecharts. After having the claims gathered and the patterns formalised in Codecharts, a visual comparison between patterns is made. Finally, a formal comparison is carried out using Codecharts and LePUS3 and Class-Z schemas as has been shown in the previous section when comparing Figure 32 and Figure 33.

An example of claims of relations is the relation between Check Point security pattern and the Strategy design pattern. The below gathered claims show that the Check Point pattern has a strong relation with the strategy pattern [9], [109]. However, using Codecharts, the claims of relations among patterns have investigated in order to validate them.

- "*This CHECK POINT interface corresponds to the abstract strategy in STRATEGY [GoF95]. The interface will provide hooks for I&A, authorization, handling unsuccessful attempts.*" [9, p. 291]
- "*CHECK POINT uses STRATEGY [GoF95] for gaining flexibility in application security*". [9, p. 296]
- "*Key feature of Check Point is that the security policy can be changed in a single place. In this way, the Check Point algorithm is a Strategy [GHJV 95].*" [109, p. 8]
- "*The Check Point algorithm uses a Strategy for application security.*"[109, p. 10]

So, now claims of relations among patterns have been gathered and the Check Point pattern formalised in Codecharts according the pattern description in the sources of the claims. The Check Point variant is shown in Figure 30. Regarding the strategy design pattern, the already formalised pattern using Codechart from [38] as shown in Figure 34 can be imported.

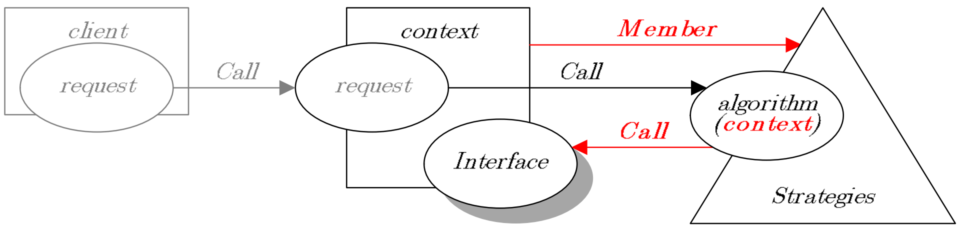


Figure 34: Codechart: Strategy pattern [38]

From the two formalised patterns in Codecharts, it can be seen clearly that Check Point does not include a complete implementation of the Strategy pattern. Figure 30 and Figure 34 show that the patterns in fact differ substantially. In particular, the

Strategy pattern as described in [18] includes at least three constraints that are missing from the Check Point pattern, highlighted in red in Figure 34:

- The *context* class has a member (attribute or field) of type *AbstractStrategy*, whereas the *SingleAccessPoint* is not expected to have a member of type *AbstractCheckPoint*.
- The set of dynamically-bound algorithm methods defined in the Strategies hierarchy call methods in the interface of context, whereas the set of dynamically-bound Check methods defined in the *CheckPointHrc* hierarchy are not expected to call methods in the *SingleAccessPoint* class
- The dynamically-bound algorithm methods defined in the Strategies hierarchy take the context as a formal argument whereas the set of dynamically-bound Check methods defined in the *CheckPointHrc* hierarchy are not expected to take *SingleAccessPoint* as a formal argument.

It is beneficial to explain the meaning of the greying out notation (*Interface*) shown in Figure 34. By looking at Codecharts vocabulary illustrated in Figure 5, it can be seen that this notation is a *1-dim signature variable* which denotes a set of methods, defined in the *Context Class,* that can be called by the dynamically-bound algorithm methods, defined in the *Strategies Hierarchy*. The *Call* relation between these two variables (*Interface* on *Context and Algorithms* on *Strategies*) is derived from the following statements in [18] in participants section of Strategy pattern description.

- "*Context*: *may define an interface that lets Strategy access its data*" [18] Page 351
- "*Context must define a more elaborate interface to its data which couples Strategy and Context more closely*" [18] Page 354

It can be concluded that the Check Point security pattern variants in [9], [109] share the idea of having a set of "Checking" strategies; however, the Strategy pattern [18] imposes more constraints than the Check Point variant in [9], [109]. This leads us to the conclusion that, on the claims shown above, the Check Point pattern does not make a full use of the Strategy pattern. Furthermore, attaching the Check the Point pattern to the Strategy pattern might mislead to incorrect implementation of the Check Point pattern when following the descriptions that claim the use of the Strategy pattern in the Check Point pattern.

An example of claims of composition relations among security patterns is the relation between the Single Access Point pattern and the Check Point pattern. A number of security patterns catalogues [9], [27], [109] claim the relation between these two patterns. The next quotes from those catalogues show the claims of the relation between the Single Access Point pattern and the Check Point pattern.

- "*CHECK POINT (287) defines the interface to be supported by concrete implementations to provide the I&A service to the SINGLE ACCESS POINT (279)*". [9, p. 289]
- "*SINGLE ACCESS POINT (279) usually calls CHECK POINT (287), providing a client's identification and the authentication information they provided.*" [9, p. 291]
- "*The Check Point Security Pattern enforces the current security policy, by monitoring the traffic passing the Single Access Point.*" [27, p. 26]
- "*Single Access Point is predestined to be combined with a Check Point.*" [27, p. 28]

- "*The implementation of Check Point combines several patterns. Single Access Point is used to ensure that security checks are performed correctly and that no initial security checks are skipped.*" [109, p. 8]

In order to test and validate the above claims, the patterns were formalised using Codecharts as shown in Figure 24, Figure 31, and Figure 30. The descriptions used to formalise the patterns are the same descriptions [9], [27], [109] where the claims of the relation are taken from. It can be seen that the aim is to validate the claims for relations of two Check Point pattern variants (Figure 30 and Figure 31) with one Single Access Point pattern variant (Figure 24). This is due to the fact that the Single Access Point variant in [9] is not complete and could not be formalised as an essential part of the pattern variant is described imprecisely using statements such as "*Protect the boundary of your system.*". Such statements combine two important aspects of the pattern. These aspects are the protection and the system boundary. It is obvious that the statement is ambiguous and imprecise especially when combined with other statements in the same description.

Having the patterns formalised in Codecharts, it can be visually noticed that the patterns have a particular participant in common called "*Client*". The Check Point variant shown in Figure 30 also has this participant; however, it has been named as "*SingleAccessPoint*", which is actually playing the same role of "*Client*" in others which are shown in Figure 24, Figure 30, and Figure 31.

With the participant in common in the patterns at hand and with the above claims of relations, it is possible to link the Check Point pattern and the Single Access Point pattern using this participant "*Client*". The results of the linkage are a new pattern which compose the two patterns as shown in Figure 35. Although this diagram demonstrates the visual linkage between the two patterns, formality is not absent as Figure 36 shows the formal representation of the Codecharts shown in Figure 35.

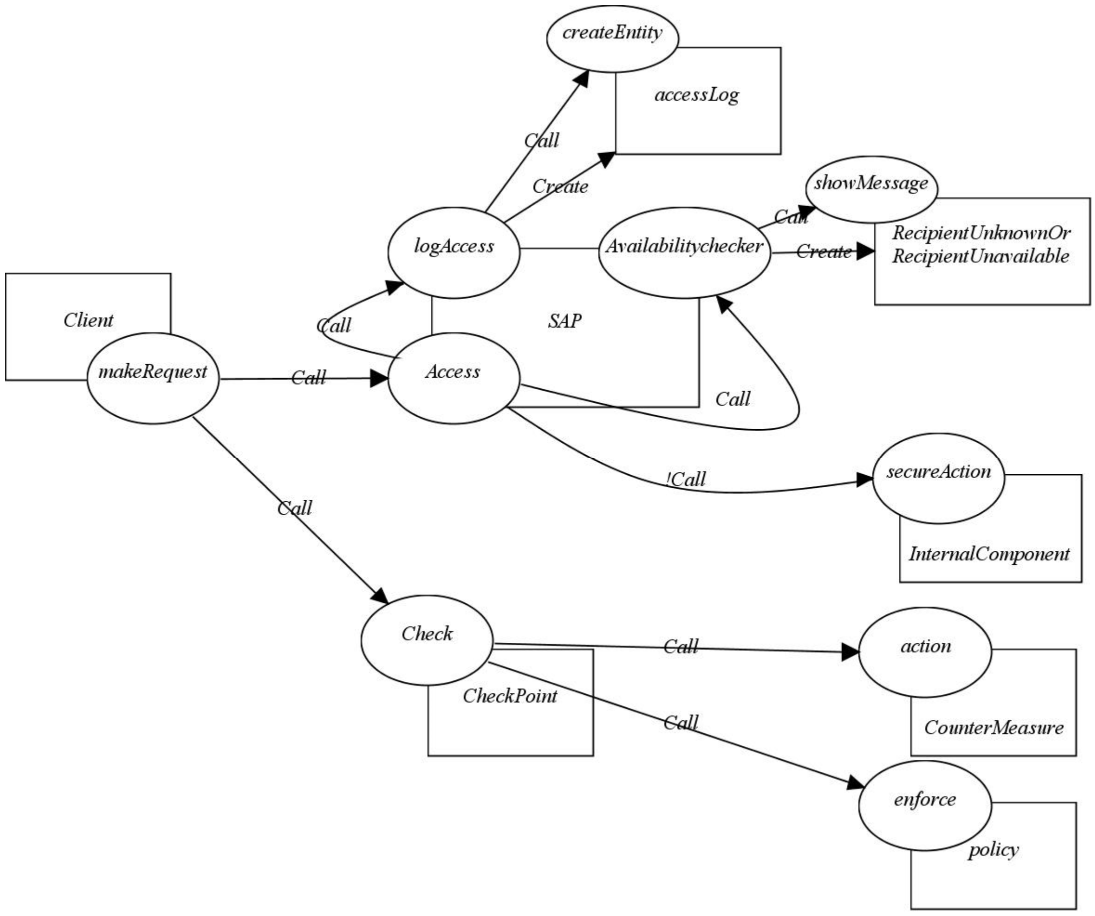


Figure 35: Codecharts: Single Access Point & Check Point patterns

```
SAP&CheckPoint
CheckPoint , Client , CounterMeasure , InternalComponent , Policy : CLASS
RecipientUnknownOrRecipientUnavailable , SAP , accessLog : CLASS
Access , Availabilitychecker , Check , action , createEntity : SIGNATURE
enforce , logAccess , makeRequest , secureAction , showMessage : SIGNATURE
--------------------------------------------------------------------------
LEFTEXCLUSIVE ( Call , Access ⊗ SAP , secureAction ⊗ InternalComponent )
Call ( Access ⊗ SAP , logAccess ⊗ SAP )
Call ( Access ⊗ SAP , Availabilitychecker ⊗ SAP )
Call ( Availabilitychecker ⊗ SAP , showMessage ⊗ RecipientUnknownOrRecipientUnavailable )
Call ( Check ⊗ CheckPoint , action ⊗ CounterMeasure )
Call ( Check ⊗ CheckPoint , enforce ⊗ Policy )
Call ( logAccess ⊗ SAP , createEntity ⊗ accessLog )
Call ( makeRequest ⊗ Client , Access ⊗ SAP )
Call ( makeRequest ⊗ Client , Check ⊗ CheckPoint )
Create ( Availabilitychecker ⊗ SAP , RecipientUnknownOrRecipientUnavailable )
Create ( logAccess ⊗ SAP , accessLog )
```

Figure 36: LePUS3 and Class-Z schema of Single Access Point & Check Point patterns

The Check Point pattern variant, shown in Figure 31, is selected to be linked with the Single Access Point pattern. The main reason for this is that the selected pattern variant separates the "*Policy*" participant from the "*CheckPoint*" participant. This itemise more on the participants of the patterns when studying the composition of patterns together. In addition, this pattern variant has been checked and verified to be implemented in an open source code. The checking of this variant is shown in the section entitled "Check Point pattern in JAAS" in this document. Moreover, the Single Access Point pattern has been checked and verified to be implemented in the same open source code. The checking of Single Access Point pattern is shown in the section entitled "Single Access Point pattern in JAAS" also in this document.

Having modelling of Figure 35 of the composition of the two patterns, the claim, which states that the two patterns are often composed together, has been put under investigation. In order to complete the investigation, the followings have been used: the Codecharts (Figure 35) of the two patterns, the source code that the patterns are proven to be implemented in separately, and our pattern detection algorithm in order to find the new pattern (Figure 35), which composed the two patterns together, and to check whether the patterns are composed together.

The main findings of our investigation are strengthening the claims of the relations between the Single Access Point pattern and the Check Point pattern. They especially strengthen the claims which show that the patterns are often composed together. This case study is shown in the section entitled “Single Access Point & Check Point in JAAS”. In addition, from the results of the case study, the claims can be evidently supported and it can be said that the Single Access Point pattern has a strong relation with the Check Point pattern as whenever the Check Point is implemented, the Single Access Point is also implemented with it.

5.3 Summary

In conclusion, this chapter has discussed the security pattern variations and the inconsistency of the descriptions of the same pattern as the cause of the variations. This has been illustrated with an example of Check Point security pattern variations in different security patterns catalogues. In addition, this chapter has discussed the relations among security patterns and with the design pattern. A number of examples of such claims of relations have been shown such as relations between the Check Point security pattern and the Strategy design pattern, and relations between the Single Access Point and the Check Point patterns. Using Codecharts, some of those claims of relations have investigated and validated.

6 Case Studies: Formalisation and manual conformance checking of security patterns

Note: some parts of this chapter were published in:

1. Alzahrani, A.A.H., A.H. Eden, and M.Z. Yafi. 2014. 'Structural Analysis of the Check Point Pattern'. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE)*, 404–8. doi:10.1109/SOSE.2014.56.
2. Alzahrani, Abdullah AH, Amnon H. Eden, and Majd Zohri Yafi. "Conformance checking of Single Access Point pattern in JAAS using Codecharts." *Information Technology and Computer Applications Congress (WCITCA), 2015 World Congress on*. IEEE, 2015..

This chapter describes a number of case studies which have been conducted to check the design conformance (design verification) of a number of security patterns in some open source java-based implementations. The case studies were manually conducted to find and check conformance of instances of security patterns in source codes that are claimed to implement the patterns. The reason behind these case studies is to evaluate the possibility of automating (and the need for automating) patterns instance finding (or pattern detection). In addition, the case studies are to test the use of Codecharts for formal modelling and conformance verification of security patterns.

In this chapter, first, the manual conformance checking (verification) process is explained. In addition, the open source codes, which are used in the case studies, are briefly described. Second, a number of case studies are shown and explained. Finally, the chapter is completed by a discussion of the results and the benefits offered by manual conformance checking of security patterns.

6.1 Manual conformance checking of security patterns

Manual conformance checking of security patterns generally refers to a human involvement to help with checking the conformance of patterns in a source code. This includes two phases. The first phase includes a manual analysis of the source code and the documentation if available. The process in this phase requires a human to study a source code (which has been claimed to implement a security pattern) line by line in order to understand the code and its components.

The second phase is manually finding code components which have relations that resemble the pattern's participants and relations. This phase includes a number of sub processes. First, after using the TTP Toolkit to extract the model of the studied source code, human interaction is needed to decide on the linkage between the source code components and the pattern participants. Second, a manual creation of assignments (mapping) between the pattern's participants and the code components will be the next. Figure 46 shows an example of these assignments which are manually created. Finally, sending the assignments to TTP Toolkit verifier manually is required in order to verify the design conformance of the pattern in the source code.

The TTP Toolkit [38]; [13] is used for model extraction of the source codes. In addition, it is used to verify the design conformance using the manually created assignments from the Codecharts of the pattern to the source code. Moreover, it has been used to visualise the model of the source code and the instances of patterns.

The TTP Toolkit is used for round-trip software engineering for OO programs. This system is developed by a software engineering team at the School of Computer Science and Electronic Engineering at the University of Essex. It supports software modelling, automated design verification (conformance checking) and design recovery.

After having the security pattern formalised using Codecharts as shown in the chapter entitled "Security Patterns Formalisation", claims stating that some well-known systems' implementations apply/implement these patterns have been traced. Some claims were found and the source code of implementations, mentioned in the claims, were searched for. Next, the TTP Toolkit has been used to extract the models for these source codes.

6.2 Open source software for the Case studies

In the following, the main open source implementation which is used in this section's case studies is briefly described and discussed. Three main open source codes are used in the case studies. These open source codes are JAAS, Log4J, and Java Apache Struts. These implementations are written in Java programming language. This allowed the case studies to be conducted as the current version on the TTP Toolkit only supports model extraction of the Java-based source code. However, the TTP Toolkit was designed to be extendable for any object-oriented programing language.

- **JAAS**

Java Authentication and Authorisation Service (JAAS) is the Java implementation of the Pluggable Authentication Module (PAM) framework originally developed for Sun's Solaris operating system. It is a standard for providing application-level security. In addition, it has been integrated as a package into Java SDK 1.4v. JAAS provides a useful pluggable framework which can be configured to offer authentication and authorisation services [26], [116], [117].

- **Log4J**

Log4j is a widely known and used Java-based logging utility written and used in Java programing languages. It was first introduced in 1996 and is freely available at http://logging.apache.org/log4j/2.x/. Log4J provides an API logging facility. In addition, it allows store logging output in a permanent place. This allows further studies of the log afterward. Log4j is designed to be reliable, simple, and easy to understand and use [118], [119].

- **Java Apache Struts**

Java Apache Struts is an MVC (Model-View-Controller) framework developed using Java programming language. In addition, it is a free and open-source framework and a part of the Apache Software Foundation project. Apache Struts allows programmers to develop contemporary web application. It provides support for the uses of REST, AJAX and JSON. The source code of Apache Struts is freely available for downloading at https://struts.apache.org/ [120], [121].

6.3 Case studies results

The following are some examples of the case studies on manual conformance checking of security patterns in some source codes which are claimed to implement the patterns. Java-based source codes were the focus in the case studies as the current version of the TTP Toolkit allows only the extraction of the model of Java-based source codes.

6.3.1 Intercepting Validator in Apache Struts

Alur et al. [26] have claimed that the Intercepting Validator pattern has been implemented in Apache Struts. As shown in the chapter entitled "Security Patterns Formalisation", the pattern has been formally modelled using Codecharts and Figure 26 illustrates the Codecharts of the pattern. With the claim of implementation and the Codecharts of the pattern, a case study of manual conformance checking of the pattern in Apache Struts was conducted and an instance was found.

Figure 37 visualises the manually found instance of the Intercepting Validator pattern. As it can be seen, the instance contains a 1-dim hierarchy constant called "validators" which was manually created. It consists of three classed in Apache Struts, namely ActionValidatorManager, DefualtActionValidatorManager, and AnnotationActionValidatorManager. Figure 38 illustrates the classes that construct the hierarchy of "validators" which has ActionValidatorManager as the superclass for the other classes.

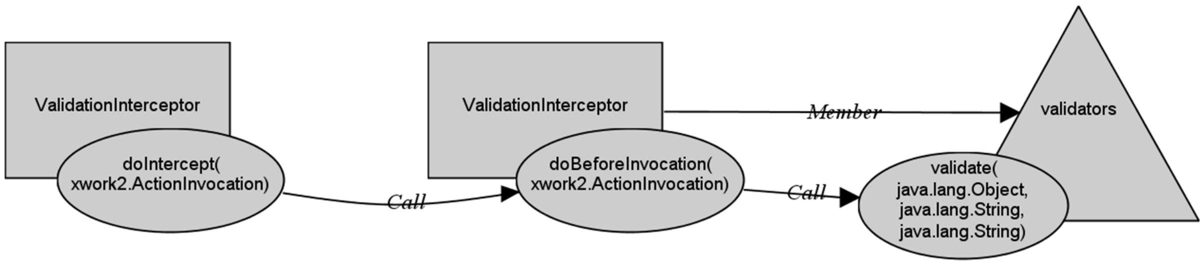
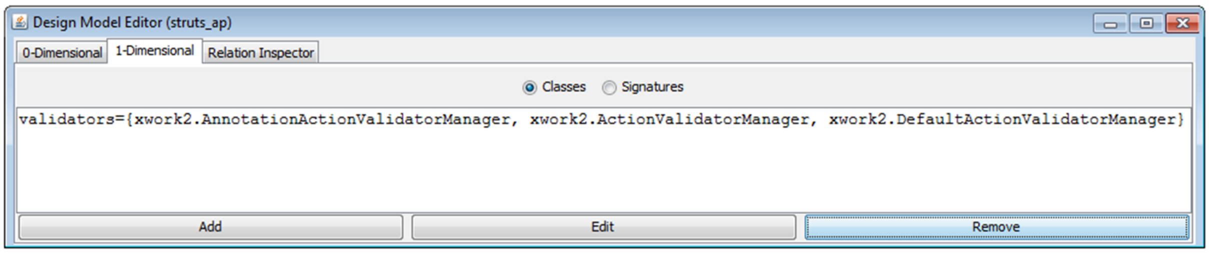

ValidationInterceptor
doIntercept(xwork2.ActionInvocation)
Call
ValidationInterceptor
doBeforeInvocation(xwork2.ActionInvocation)
Member
Call
validators
validate(java.lang.Object, java.lang.String, java.lang.String)

Figure 37: Codechart: An instance of Intercepting Validator Pattern in Apache Struts

Looking at the instance in Figure 37, the 1-dim hierarchy constant called "validators" practically demonstrates the benefits of using Codecharts to model the abstraction in patterns as "validators" can have any number of subclasses of the superclass ActionValidatorManager. In addition, it can be noticed that the class ValidationInterceptor is playing the role of two participants in the pattern, namely SecureActionBase and InterceptingValidator. This ability allows finding the instances of a pattern when some roles of the pattern's participants are encapsulated in one component in the source code.

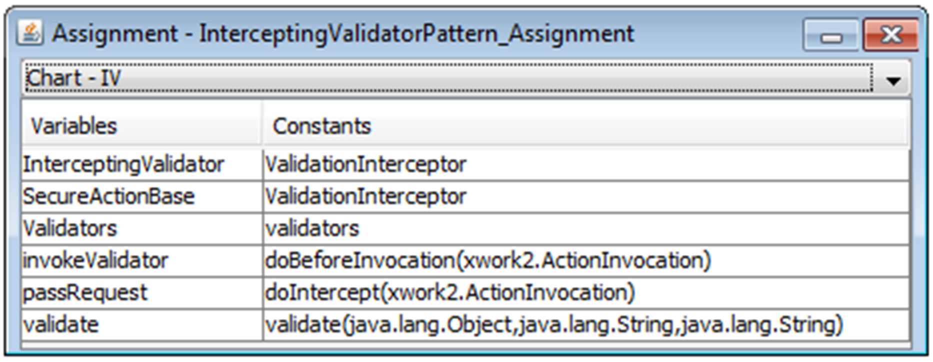


[✓ Classes] Signatures

```
validators={xwork2.AnnotationActionValidatorManager, xwork2.ActionValidatorManager, xwork2.DefaultActionValidatorManager}
```

Figure 38: Manually-defined higher-dimension entities (hierarchy) (Intercepting Validator - Apache Struts)

In order to verify the design conformance of the instance (Figure 37) found in Apache Struts of the Intercepting Validator pattern, an assignment from the pattern to the source code components was manually created in the TTP Toolkit. Figure 39 shows this assignment (mapping). Then the TTP Toolkit verifier was used to verify the instance and the assignment. Figure 40 illustrates the results from the TTP Toolkit verifier. As can be seen in Figure 40, both the instance (Figure 37) and the assignment (Figure 39) have passed the design verification according to the TTP Toolkit verifier.

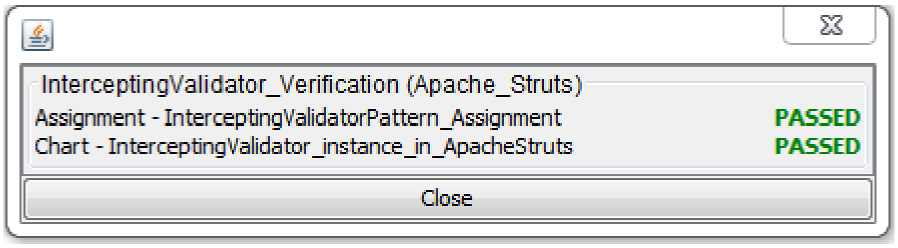


Assignment - InterceptingValidatorPattern_Assignment

Chart - IV

Variables	Constants
InterceptingValidator	ValidationInterceptor
SecureActionBase	ValidationInterceptor
Validators	validators
invokeValidator	doBeforeInvocation(xwork2.ActionInvocation)
passRequest	doIntercept(xwork2.ActionInvocation)
validate	validate(java.lang.Object,java.lang.String,java.lang.String)

Figure 39: Manually created assignment of Intercepting Validator Pattern in Apache Struts



```
InterceptingValidator_Verification (Apache_Struts)
Assignment - InterceptingValidatorPattern_Assignment        PASSED
Chart - InterceptingValidator_instance_in_ApacheStruts      PASSED
```

Figure 40: Verification Result of Intercepting Validator Pattern instance in Apache Struts

6.3.2 Secure Logger in Java Logging API

Alur et al. [26], in their catalogue, claimed that the Secure Logger is implemented in Java Logging API (Log4J). In addition, in this research, the same catalogue was used to formally model the structure of the pattern. However, the Secure Logger case study is a special case among all case studies. It was more appropriate to articulate the result as “Partially-Passed”. In the source code, the pattern was partially implemented. This was due to the absence of the participant (Figure 41) called “Secure Logger” and superimposition on it called “Log”.

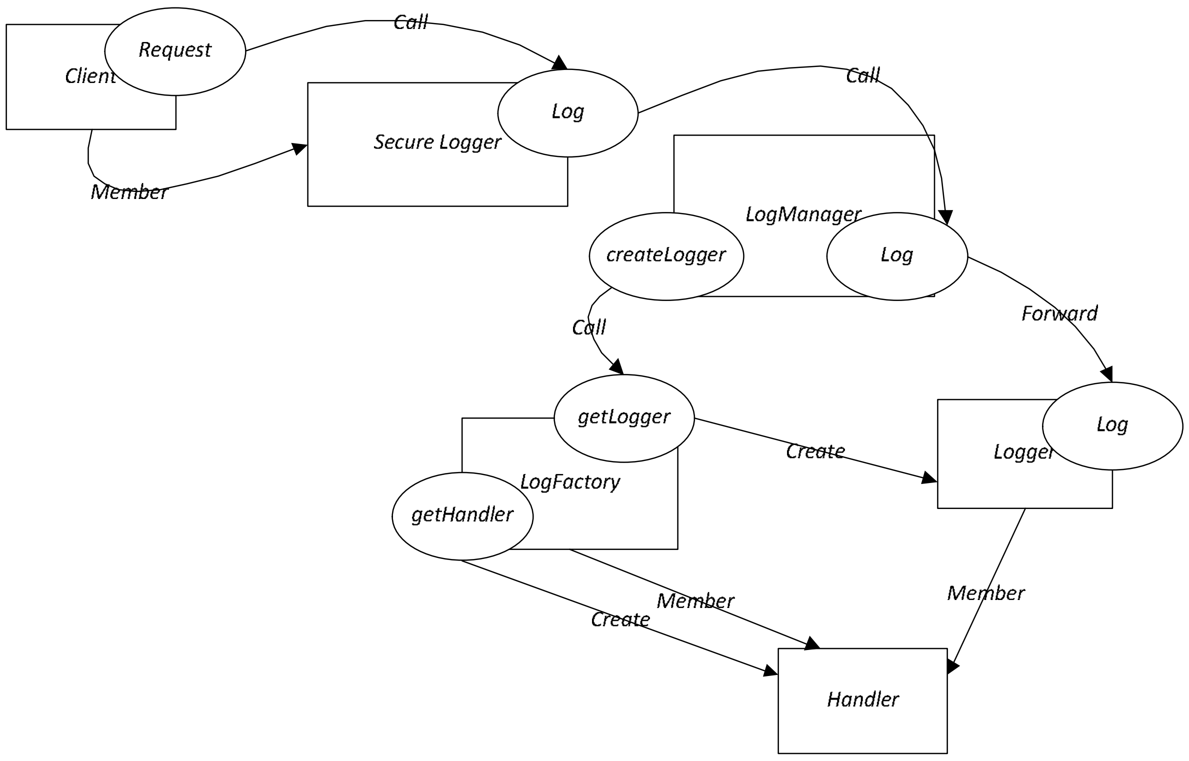


Figure 41: Codecharts: Secure Logger pattern

Apart from the absence of "Secure Logger", all the participants, superimpositions, and relations were implemented and could be found in the source code as shown in Figure 42. Regarding the "Create" relation shown in the pattern (Figure 41) from "getHandler" to "Handler", the "Produce" relation in the instance (Figure 42) is reflecting the same meaning as any "Produce" relation between two components in the source code (getHandler and Handler[]) has to be, first, in the "Create" relation [108]. Therefore, this relation is satisfied in the instance found in the pattern.

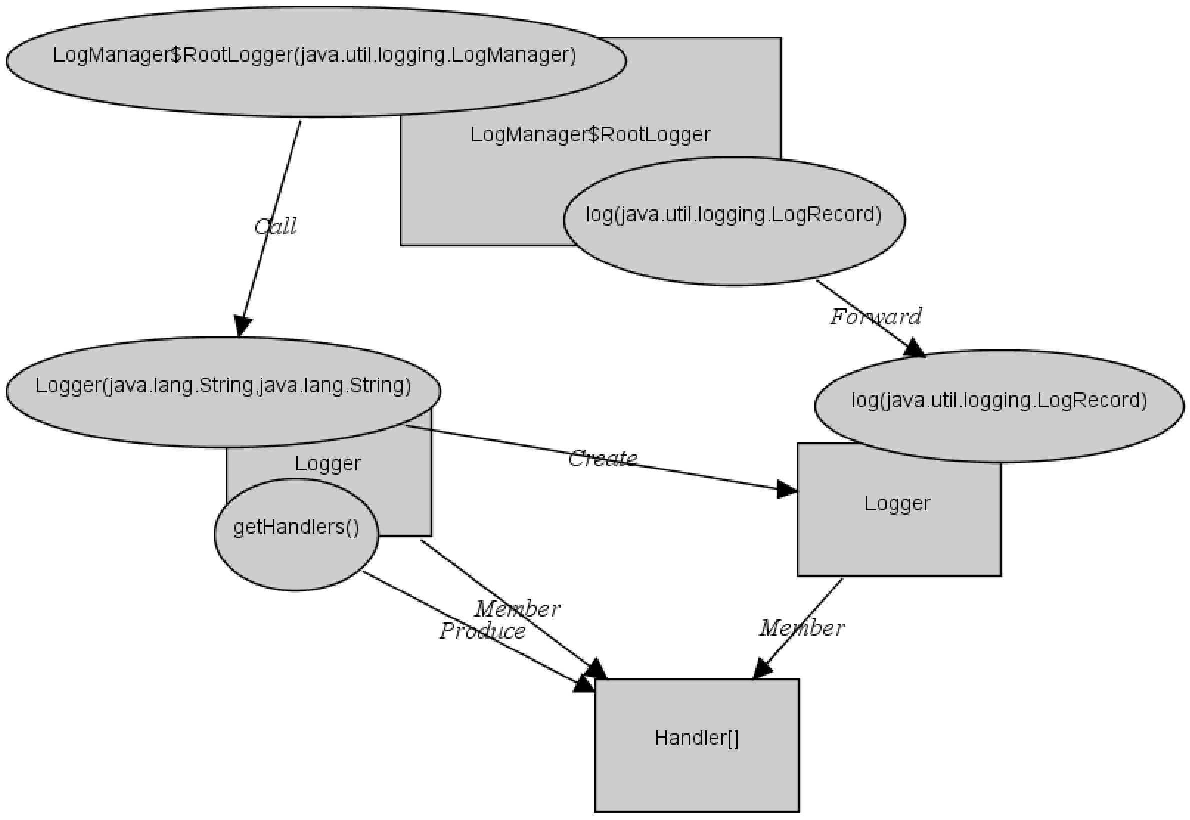
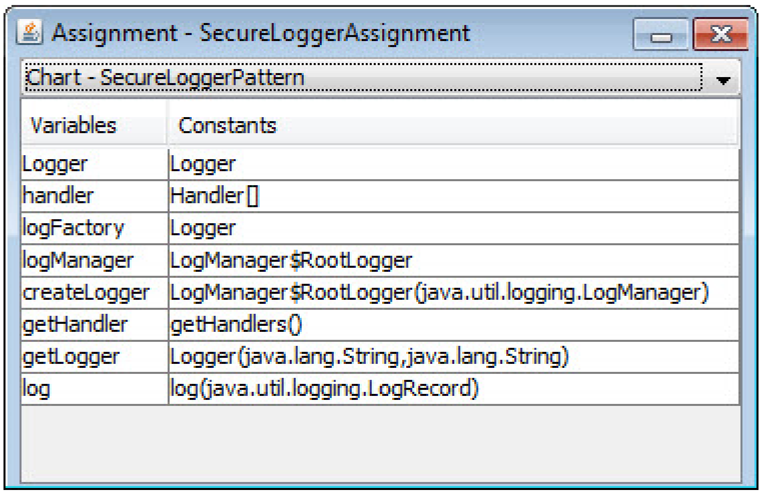

LogManager$RootLogger(java.util.logging.LogManager)
LogManager$RootLogger
log(java.util.logging.LogRecord)
Call
Forward
Logger(java.lang.String,java.lang.String)
log(java.util.logging.LogRecord)
Logger
Create
Logger
getHandlers()
Member
Produce
Member
Handler[]

Figure 42: Codechart: An instance of Secure Logger Pattern in Java Logging API

When the manual analysis of the Java Logging API (Log4J) was completed and an incomplete instance of the pattern was found, an assignment (mapping) from the pattern to the source code components was manually created using the TTP Toolkit as shown in Figure 43. This was for design conformance verification of the detected instance. The TTP Toolkit verifier, then, was used to check the detected instance conformance to parts of the pattern. These parts were missing the participants “Client”, “Secure Logger”, superimpositions of “Request” over “Client”, superimpositions of “Log” over “Secure Logger”, and the relation among them as shown in Figure 41.



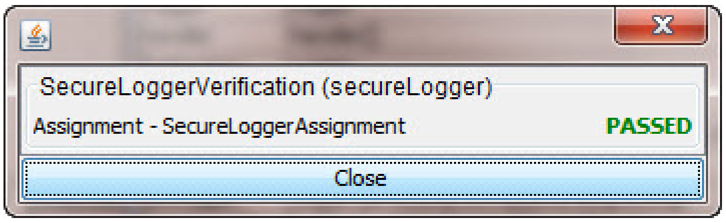
Assignment - SecureLoggerAssignment

Chart - SecureLoggerPattern

Variables	Constants
Logger	Logger
handler	Handler[]
logFactory	Logger
logManager	LogManager$RootLogger
createLogger	LogManager$RootLogger(java.util.logging.LogManager)
getHandler	getHandlers()
getLogger	Logger(java.lang.String,java.lang.String)
log	log(java.util.logging.LogRecord)

Figure 43: Manually created assignment of Secure Logger Pattern in Java Logging API

The results of verifying the assignment, Figure 43, from the pattern to the source code components was "Passed" as can be seen in Figure 44. However, as explained earlier, the assignment was to map some participants of the pattern to components in the source code. Therefore, the pattern is not implemented completely in Log4J. So, it is concluded that the pattern is "Partially-implemented" in Log4J.



SecureLoggerVerification (secureLogger)

Assignment - SecureLoggerAssignment PASSED

Figure 44: Verification Result of Secure Logger Pattern instance in Java Logging API

6.3.3 Single Access Point pattern in JAAS

This case study shows an investigation of an implementation of a Single Access Point (SAP) in JAAS. The case study is to search for an instance of an implementation of the pattern in the source code of JAAS. Many researchers [9], [26], [106] claimed that SAP is implemented in Login services which JAAS is one of and/or is used in. With the claim, the source code of JAAS [116], and the SAP pattern Codecharts (Figure 24), a manual search for an instance of SAP in JAAS was carried out.

The main feature in SAP Codecharts is the modelling of the secured component. This is done using the Exclusive Operator (Exclamation symbol) on the relations with secured component or participants. This has been discussed in the section entitled "Single Access Point pattern in Codecharts" in "Security Patterns Formalisation" chapter. This feature allows controlling the access and allow only one component of the code to access the secured component.

Figure 45 shows the detected instance of SAP in JAAS. It can be noticed that the role of secureAction (which is a component whose access should be controlled) is played by "Subject()" in the class "Subject". This means that there is no method in any class of JAAS calling "Subject()" in the class "Subject" except "login()" in the class "LoginContext".

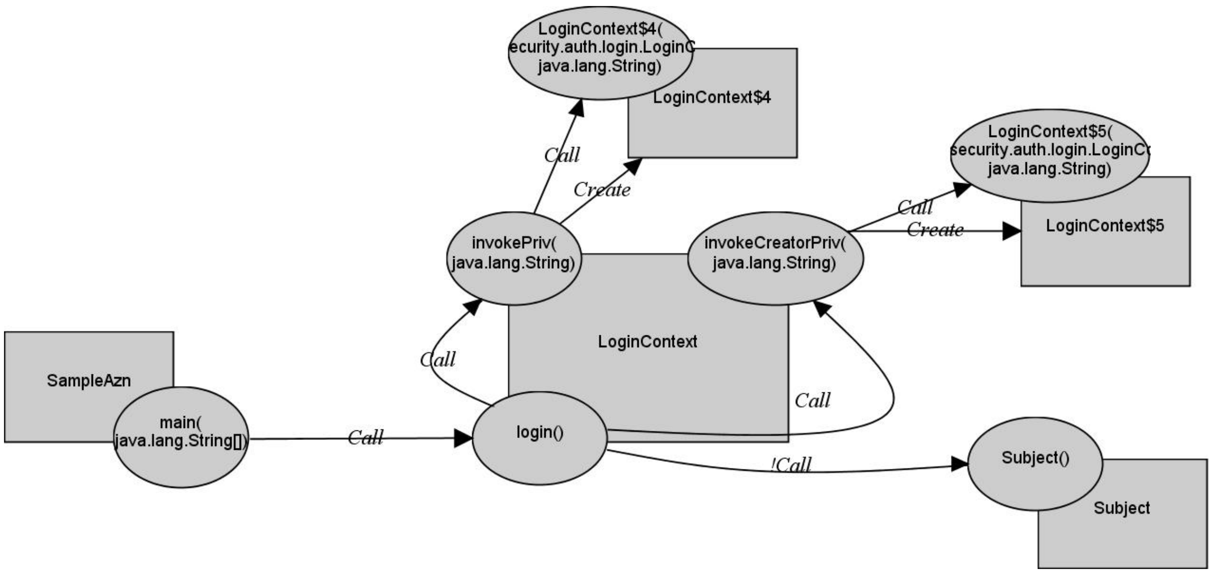


Figure 45: Codechart: An instance of SAP in JAAS

In order to verify the detected instance (which uses Exclusive Operator) of SAP in JAAS, the TTP Toolkit was used to manually create assignment (mapping) from the SAP Pattern to the source code components. Figure 46 illustrates this assignment, while Figure 47 shows the verification results of the TTP Toolkit verifier of the instance and the assignment shown in Figure 45 and Figure 46, respectively.

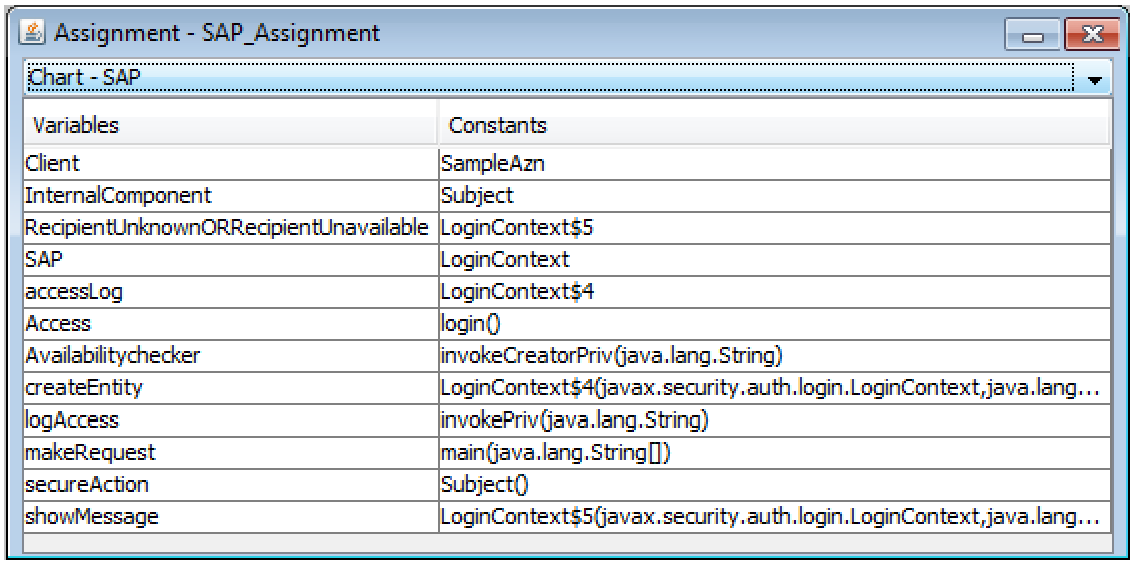
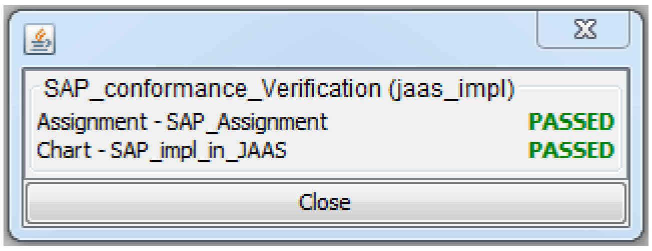

Assignment - SAP_Assignment

Chart - SAP

Variables	Constants
Client	SampleAzn
InternalComponent	Subject
RecipientUnknownORRecipientUnavailable	LoginContext$5
SAP	LoginContext
accessLog	LoginContext$4
Access	login()
Availabilitychecker	invokeCreatorPriv(java.lang.String)
createEntity	LoginContext$4(javax.security.auth.login.LoginContext,java.lang...
logAccess	invokePriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	Subject()
showMessage	LoginContext$5(javax.security.auth.login.LoginContext,java.lang...

Figure 46: Manually created assignment of SAP in JAAS



SAP_conformance_Verification (jaas_impl)

```
Assignment - SAP_Assignment        PASSED
Chart - SAP_impl_in_JAAS           PASSED
```

Figure 47: Verification Result of SAP instance in JAAS

The verification results of the TTP Toolkit verifier shown in Figure 47 demonstrate that the instance found in JAAS passed the design conformance check against the SAP specifications in the Codecharts shown in Figure 24. This includes a check and assurance of that, in JAAS source code, only "login()" in the class "LoginContext" is calling "Subject()" in the class "Subject".

6.4 Discussion

It can be seen from Figure 37, Figure 42, and Figure 45 that instances of the investigated patterns are found. Moreover, these instances are verified using the TTP Toolkit verifier as illustrated in Figure 40, Figure 44, and Figure 47. The verifications for the patterns were carried out using the Assignments (mappings) from a pattern to the source code components. These assignments are illustrated in Figure 39, Figure 43, and Figure 46.

In this chapter, a number of case studies on manual design conformance checking of security patterns were shown. However, for more case studies, APPENDIX B shows more results. In order to summarise the above case studies and other case studies (APPENDIX B), Table 6 indicates the main findings and results of the case studies. It is obvious from the table that for each pattern an instance is found in the searched source code. Furthermore, these instances were verified and they passed the design conformance verification.

Table 6: Security Patterns Design Conformance Checking results

Pattern \ Open source	JAAS	Apache Struts	Java Logging (API) – Log4J
SAP	Passed	-	-
Check Point	Passed	-	-
Secure Architecture	Passed	-	-
Secure Logger	-	-	Partially-Passed
Intercepting Validator	-	Passed	-

As the case studies were manually conducted to find instances of the patterns from the source code, the time spent was important to calculate. Therefore, Table 7 illustrates the time spent on each case study. The time is shown in hours and explains time is spent on reading the source code line by line and the documentations of the source code when available, as well as exploring the extracted model of the source code. It is obvious that 115 hours makes around 5 days. This shows the extensive effort to find an instance of each security pattern under consideration in a source code. Consequently, a motivation to automate this process is generated.

Table 7: Time spent on case studies

Pattern \ Open source	JAAS	Apache Struts	Java Logging (API) – Log4J
SAP	20 hours	-	-
Check Point	12 hours	-	-
Secure Architecture	48 hours	-	-
Secure Logger	-	-	16 hours
Intercepting Validator	-	19 hours	-
Total time	**115** hours		

With the large amount of time taken to conduct the case studies, one instance is found in each pattern case study. Although the results are sought after, they lack in comprehension. This is due to the obvious investigation question which could be asked. The question is "*Is this instance the only instance of a pattern in the given source code?*". The case studies do not answer this question. Moreover, in order to answer this question, more time is needed for studying the source code.

However, the manual design conformance of security patterns offers a number of benefits, one of which is the ability of finding incomplete implementation of patterns. This is demonstrated in the case study of Secure Logger. Another benefit is recognising the component (in the source code) that plays the roles of more than one participant of a pattern. This is shown in the case studies of the Intercepting Validator pattern and Secure Logger.

6.5 Summary

In conclusion, these case studies consume a considerable amount of time as they need a full manual analysis of the source code. Furthermore, they do not show all the instances of a pattern in a source code. These two issues (time and number of instances) are the motivation behind this research's second direction, which is pattern detection. Furthermore, the case studies practically showed the needs of automatic approach for finding instances of a given pattern in a given source code as well as for automatically generating the mappings from the patterns' participants to the source code components. In addition, as using the TTP Toolkit was manual, the need of automatic use emerges. This last thing is to complete the circle of automatic pattern finding. So, the second direction of this research is to automatically search for a given pattern in a given source code, automatically generate the mappings (Assignments) from the pattern to instances, and automatically verify the instances design conformance using the TTP Toolkit.

7 Pattern Detection

This chapter will discuss the pattern detection theoretical side. The section will begin with a clear problem definition statement. Next, all the terminologies and notational conventions which will be used will be clarified. After this, the brute force solution for the pattern detection problem will be shown in the respective section.

In addition, this chapter will show our pattern detection solution and discuss the notion of minimising the search space in the source code. Our solution will be shown as an algorithm. Moreover, the theoretical comparison analysis between the pattern detection solutions will be made and explained.

Furthermore, in this chapter, the proof of our pattern detection algorithm will be shown and explained. Finally, the implementation of our pattern detection algorithm will elaborated. This includes illustrating the integration of it in the TTP Toolkit.

7.1 Terminology

1. **Specification** Ψ (open specification) is on a par with a set of open formulas f in the predicate calculus [38, p. 183].
2. **Design model** [38, p. 230] LePUS3 Definition VII. is a triple $\mathfrak{M} = \{\mathbb{U}_*, \mathbb{R}, \mathrm{I}\}$ *where*
    - $\mathbb{U}_* \triangleq \mathbb{U}_0 \uplus \mathbb{U}_1 \uplus .... \uplus \mathbb{U}_\mathrm{d}$ *is the* ***universe*** *of* $\mathfrak{M}$ *where each* $\mathbb{U}_\mathrm{k}$ *is a finite set of entities of dimension k and d is some small natural number (usually no greater than 3)*
    - $\mathbb{R}$ *is a set of relations (Definition II) including the unary relations Class, Method, Signature, and Abstract and the binary relations Inherit, Member, Produce, Call, Return, Forward, and SignatureOf.*
    - I *is an* ***interpretation function*** *which maps some constant terms (Definition VI) to entities in* $\mathbb{U}_*$.
3. An **assignment** g from Ψ into a $\mathfrak{M}$ is a function g mapping each free variable in Ψ to a constant in $\mathrm{I}_{\mathrm{domain}}$ [38, p. 232] LePUS3 Definition XIV.
4. $\mathrm{I}_{\mathrm{domain}}$ is a domain of interpretation function [38, p. 230] LePUS3 Definition VII.

7.2 Notational conventions

1. Φ[x/v] stands for the consistent replacement of free variable v *in* Φ *with* x
2. $dim(x)$ *is the dimension of* x
3. $e_1, e_2, \ldots$ entities
4. $v_1, v_2, \ldots$ variables
5. $\mathrm{V}_\Psi := \{v_1, v_2, \ldots, v_k\} | \{v_i\}_{i=1\ldots k}$ is free variables in some $f \in \Psi$
6. *We say* $\mathfrak{M}$ *satisfies* Ψ, *written* $\mathfrak{M} \vDash \Psi$, *iff there exists some mapping* g *such that* $\mathfrak{M} \vDash_{\mathrm{g}} \Psi$
7. $$f \overset{\text{def}}{:=} \begin{cases} UnaryRelation(Domain) \\ BinaryRelation(Domain, Range) \\ ALL(UnaryRelation, Domain) \\ TOTAL(BinaryRelation, Domain, Range) \\ ISOMORPHIC(BinaryRelation, Domain, Range) \end{cases}$$
8. $\langle l_1 : x_1, .. l_k : x_k \rangle$ stands for a labelled tuple.
9. Given $r = \langle \alpha_1 : x_1, .. \alpha_n : x_n \rangle$ and $c = \langle \beta_1 : y_1, .. \beta_m : y_m \rangle$ $then$
    - $r \cdot c := \langle \alpha_1 : x_1, .. \alpha_n : x_n, \beta_1 : y_1, .. \beta_m : y_m \rangle$

      and
    - $r.\alpha_i := \langle \alpha_i : x_i \rangle$ $where$ $1 \leq i \geq$

      and
    - $r[\alpha_i] := x_i$

      and
    - $r = c$ iff $n = m$ and $\{\alpha_i = \beta_i\}$ and $\{x_i = y_i\}$ for $i = 1 \ldots n$
    - $L_r := \{\alpha_i\}_{i=0 \to n}$
10. Given $Y := \{r\}$ $then$ $L_Y = L_r$
11. $g : \mathrm{V}_\Psi \to \{c[v_i]\}_{i=1\ldots|\mathrm{V}_\Psi|}$ $such$ $that$ $g(v_i) := c[v_i]$
12. $\mathrm{V}_\Psi^0 := \{v \in \mathrm{V}_\Psi | \dim(v) = 0\}$

13. $\mathrm{V}_{\Psi}^{1} := \{v \in \mathrm{V}_{\Psi} \mid \dim(v) = 1\}$

Notes:

- Definition 9 is to describe the handling of the labelled tuples in different processes such as joining labelled tuples by checking the values which has the same labels, retrieving the value in a tuple with its label.
- Definition 10 shows that if $\{r\}$ is a set labelled tuples and has been copied into Y then L_Y (which is the set of the labels) will be the same of the L_r .
- Definition 11 is showing the mean of assignment g that is used to replace each variable v in the specification Ψ with a found entity c from the model of the source code.
- Definition 12 and 13 describe the sets that will contain the free variables shown in the according to their dimensions as the variables of dim-1 are the notations of set of classes, signatures, and hierarchies, whereas, the variables of dim-0 are the notations of classes and signatures.

7.3 Brute force pattern detection algorithm and complexity

In this section, Table 8 shows the brute force solution for the problem of pattern detection, which has been defined in the introduction section of this document. Next, the inputs and the expected outputs of pattern detection are described and elaborated. In addition, Table 8 illustrates the complexity study of this algorithm.

- Input: Ψ, $\mathfrak{M}$
- Output: a set of assignments $\{g\}$ mapping variables from Ψ to constants in $\mathfrak{M}$ such that $\mathfrak{M} \vDash_{\mathrm{g}} \Psi$
- $w = |\mathbb{U}_0| + |\mathcal{P}(\mathbb{U}_0)|$, *is the number of 0-dim and 1-dim entities in* $\mathfrak{M}$ where $|\mathcal{P}(\mathbb{U}_0)|$ *is the number of all possible 1-dim entities*
- $n = |\mathrm{V}_\Psi|, n\ is\ the\ number\ of\ free\ variables\ (v)\ in\ \Psi,$ $where\ \mathrm{V}_\Psi\ has\ been\ defined\ earlier$

Table 8: Brute force pattern detection algorithm

1	$C := \{\}$	1
2.	Foreach $e \in \mathbb{U}_0 \cup \mathcal{P}(\mathbb{U}_0)$	$\mathcal{w}$
3.	$\mathbb{U}_{\mathrm{dim(e)}} := \mathbb{U}_{\mathrm{dim(e)}} \cup \{\mathrm{e}\}$	$\mathcal{w}$
4.	$\mathrm{I} := \mathrm{I} \cup \{\langle \mathrm{c_e}, \mathrm{e} \rangle\}$	$\mathcal{w}$
5.	$C := C \cup \{\mathrm{c_e}\}$	$\mathcal{w}$
6.	Hcaerof	
7.	$result := \{\}$	1
8.	Foreach $\vec{x} \in C^{\|V_\Psi\|}$	$\mathcal{w}^n$
9.	$\mathrm{i} := 0$	$\mathcal{w}^n$
10.	Foreach $v \in \mathrm{V}_\Psi$	$n \cdot \mathcal{w}^n$
11.	$g := g \cup \{(v, \vec{x} \downarrow i)\}$ // down arrow denote indexing	$n \cdot \mathcal{w}^n$
12.	$\mathrm{i} := \mathrm{i} + 1$	$n \cdot \mathcal{w}^n$
13.	Hcaerof	
14.	If $\mathfrak{M} \vDash_{\mathrm{g}} \Psi$[1] then $result := result \cup \{g\}$	$\mathcal{w}^n$
15.	Hcaerof	
16.	**Return** $result$	1

Complexity: $O\big(T(n, \mathcal{w})\big) = n * \mathcal{w}^n$

[1] The complexity of this step constitutes a separate topic of research

7.4 Controlling search space in proposed pattern detection algorithm

Pattern detection is a process which consists of having a pattern formalised in any language and a model of a source code. Then, the process is to map the components from the source code to the pattern's components (participants). Finally, the mapping needs to be checked where the successful cases form the pattern's instances and the failure cases are reasoned. This is done by passing all mappings to a verifier. In this research the detection inputs are a pattern formally modelled in Codecharts and a model extracted from a java source code using the TTP Toolkit [13], [38]. In addition, in order to check the mappings, the verifier used is also a TTP Toolkit verifier.

Detecting a pattern in a source code requires looking at the model of a source code to find all components and relations among them, which satisfy the given pattern formalisation and constraints. The brute force algorithm (Table 8) is one solution which simply makes all mappings combinations of components from the model of the source code to the pattern's components (participants). Then, it passes mappings to the verifier for checking and returns the successfully passed mappings.

Here is an explanation of how the proposed pattern detection algorithm (Table 9) controls the search space and attempt to look at the minimum information in order to detect a pattern's instances. However, it is important to recap some of the mathematical notations which will be used in the following explanations. The next are the notations and their descriptions:

- $\mathfrak{M}$ is the model of the source code
- $\mathbb{U}_0 \in \mathfrak{M}$ is a set of all classes and methods (entities) of the source code

- $R \in \mathbb{R}$ is a set of tuples each denotes two entities (classes and/or methods) in specific relation (e.g *Inherit*)
- $\mathbb{R} \in \mathfrak{M}$ is a set of all sets of relations between classes and methods in the source code.
- $\Psi \overset{\text{def}}{=} \{f\}$ where Ψ is the pattern formalised in LePUS3 and f is a formula.

When a pattern specification has more participants (variables) and a model of a source code consists of large number of entities, it is obvious that the brute force algorithm (Table 8) will generate all mappings combinations from the variables in the specification to entities in the model of the source code. This will be a significant number of mappings to be checked by the verifier of the TTP Toolkit, especially, in the case of a specification that has variables of dim-1 (set of methods, set of classes, or/and hierarchies). In such a case, the brute force algorithm will generate the power set of the set of entities in the model of the source code. Next, all the generated subsets will play the role of the candidates in the combinations of each dim-1 variable in the specification.

Therefore, the brute force algorithm solution is not practical. So, the need for a cleverer solution emerges. In our proposed algorithm (Table 9), the aim is to control the search space by means of considering the patterns relations (formulas) and looking only at the related components of the source code.

In order to do so, the proposed algorithm considers only $\mathbb{R}$ in the model. This allows it to have only the related entities in the mappings. Therefore, the algorithm reduces the number of mappings by eliminating the entities which are not related to pattern's constraints (steps 6, 12). Moreover, $\mathbb{R}$ is the actual set which the TTP toolkit verifier considers in the process of checking the mappings. So, the mappings from the proposed algorithm are closer to the correct instances.

Furthermore, the algorithm reduces more numbers of mappings by only considering the **corresponding** $R \in \mathbb{R}$. As described in the algorithm correctness section, when saying R corresponds to f, this means that if $f := Inherit(v_1, v_2)$ then the **corresponding** R is $\underline{Inherit} \in \mathbb{R}$. So, the algorithm retrieves a subset of $\mathbb{R}$. As a result, more entities are omitted and the only relevant entities are looked at in detecting the algorithm (steps 3, 5, 6, 11, 12 - (Table 9)).

In order to minimise the search space more and look at the only related entities, the algorithm connects the retrieved subset of $\mathbb{R}$. Connecting the subset of $\mathbb{R}$ allows the algorithm to limit the number of entities only to the entities with high potential for the pattern's participants. In order to connect the subset of $\mathbb{R}$, the algorithm labels the tuples in each subset of $\mathbb{R}$ with the names of variables in the corresponding formula f (steps 6, 12 - (Table 9)). After connecting the subsets, the algorithm uses the labels to check the connection by the names of the entities and the labels in the **joinAlgorithms** (**BinaryRelationJoin** steps 5,9,13 Table 10 —**UnaryRelationJoin** steps 4 Table 11), where the algorithms do the final eliminations of irrelevant entities and produce the mappings which are to be sent to the TTP Toolkit verifier.

As connecting the subset of $\mathbb{R}$ is one of the ways the algorithm uses in minimising the search space, the checking of connections can go through the problem of Cartesian product in the case of checking two labelled sets of the considered subset of $\mathbb{R}$ in the **joinAlgorithms (**Table 10 and Table 11**)**. Therefore, the algorithm is designed to avoid such a problem by checking whether the two sets intersect in the labels of tuples. In the case of not having common labels in the two sets, the algorithm delays the checking for the coming new set and retrieves a new set. In the case of having common labels, the algorithm carries out the checking by passing the two sets to the **joinAlgorithms** (Table 10 and Table 11**)** and removing the checked sets coming from the waiting set denoted by (σ, steps 4 Table 9). This will allow the algorithm to avoid going through the Cartesian product issue.

Regarding the patterns with participants as higher-dimensional (dim-1) variable such as set of signatures and/or set of classes, the brute force algorithm (Table 8) generates all possible dim-1 entities which can be used to replace the pattern's participants of higher-dimension. This is done by means of a power set of $\mathbb{U}_\mathbf{0}$ denoted by$\boldsymbol{\mathcal{P}}(\mathbb{U}_\mathbf{0})$. Then, it uses them to make the mappings. If a set $\mathbb{U}_\mathbf{0}$ is known to a subset the number of combinations it has can be exponential. Moreover, this would increase the search space dramatically, especially in the large scale source code. Therefore, a solution is required to avoid such an issue in the brute force algorithm (Table 8).

The proposed algorithm (Table 9) aims to avoid such an issue by providing only the ultimate candidate for the dim-1 variables (pattern's participant). So, the algorithm tries to find only the set of entities which satisfies the corresponding pattern's participant in combination with other entities satisfying the pattern's other participants. The following explains the ultimate candidate:

Let's say e is an ultimate candidate for v in a mapping g then:

$$e := \{c \mid c \in \mathbb{U}_0\} \text{ and } e \in \mathbb{U}_1 \text{ and } \mathfrak{M} \vDash_{g(e/v)} \Psi \quad \Rightarrow \quad \forall x \in \mathcal{P}(e) \colon \mathfrak{M} \vDash_{g(x/v)} \Psi$$

Finding the ultimate candidate for a higher-dimensional variable will limit the search space and allow for minimising the uncompleted pattern instance by omitting the mappings which can be considered as subsets of other mappings and produce only the complete mappings. **GroupingAlgorithm** (Table 12) is responsible of checking whether a mapping is subsets of another mapping and it combines the mappings to have in them only the ultimate candidates for the pattern's higher-dimensional participants (variables). **GroupingAlgorithm** (Table 12) uses the dim-0 pattern's participant in order to check the mappings obtained from **joinAlgorithms (**Table 10 and Table 11**)** and to cluster them depending on the pattern's participant of dim-0. Then, it produces only mappings with ultimate candidates for the pattern's higher-dimensional participants (variables).

In conclusion, the proposed pattern detection algorithm (Table 9) controls the search space in the model of a source code by considering the $\mathbb{R}$ instead of $\mathbb{U}_0$. Then, it considers the corresponding $R \in \mathbb{R}$, which represents only the relevant subset of $\mathbb{R}$. By doing so, it eliminates the entities in the source code that are irrelevant to be candidates for a given pattern's participants. Next, it connects the considered subset of $\mathbb{R}$ by means of labelling the tuples in the subsets of $\mathbb{R}$ with the variables names (pattern's participants' names) of the corresponding formula for each R in the considered subset. After connecting the sets, the algorithm starts to omit entities by checking the names of entities and, in every connected set, using the labels. The algorithm, in addition, avoids going through a Cartesian product, when checking the entities names by labels, by delaying checking the coming set R when there is no common label with the previously checked R (s). Finally, the algorithm uses the idea of the ultimate higher-dimensional candidate for the higher dimensional pattern's participants. This allows the algorithm to avoid having the solution of the power set, which increases the search space dramatically. In addition, it omits the mappings which can be regarded as subsets of other mappings.

7.5 Proposed efficient pattern detection algorithm and complexity

- Input: Ψ, $\mathfrak{M}$
- Output: a set of assignments $\{g\}$ mapping variables from Ψ to constants in $\mathfrak{M}$ such that $\mathfrak{M} \vDash_g \Psi$
- $s = |\Psi|$
- $m = max(|R|)|R \in \mathbb{R}$
- $k = |V_\Psi^1|$

Table 9: Proposed efficient pattern detection algorithm – Main

```
1.  $\beta := \{\}$                                                                         1
2.  $\sigma := \{f | f \in \Psi\}$
    While $\sigma \neq \emptyset$
3.  foreach $f \in \sigma$                                                                  $s$
4.
5.      if $f = \begin{cases} ISOMORPHIC(BinaryRelation, Domain, Range) \\ TOTAL(BinaryRelation, Domain, Range) \\ BinaryRelation(Domain, Range) \end{cases}$ then        $s$
6.          $\pi \stackrel{\text{def}}{=} \{\langle Domain: x, Range: y\rangle \mid \langle x, y\rangle \in \underline{\text{BinaryRelation}}\}$        $s$
7.          if $(L_\pi \cap L_\beta) \neq \emptyset$ then
8.              $\beta := BinaryRelationJoin(\beta, \pi, Domain, Range)$
9.              $\sigma := \sigma - \{f\}$
10.         Endif
11.     else if $f = \begin{cases} UnaryRelation(Domain) \\ ALL(UnaryRelation, Domain) \end{cases}$ then
12.         $\pi \stackrel{\text{def}}{=} \{\langle Domain: x\rangle \mid \langle x\rangle \in \underline{\text{UnaryRelation}}\}$
```

```
13.                 if $(L_\pi \cap L_\beta) \neq \emptyset$ then                 $s * m^2$
14.                     $\beta := UnaryRelationJoin(\beta, \pi, Domain)$          $s$
15.                     $\sigma := \sigma - \{f\}$                                $s$
16.                 Endif                                                         $s * m^2$
17.             Endif
18.             if $\beta = \{\}$ then
19.                 $\beta := \pi$
20.                 $\sigma := \sigma - \{f\}$
21.             Endif
22.
23.     Hcaerof
24.     End while
25.     $\beta := GroupingAlgorithm(\beta, V_\Psi^0, V_\Psi^1)$                   $k * m^2$
26.     $result := \{\}$
27.     foreach $c \in \beta$                                                     $m^2$
28.         $g: V_\Psi \rightarrow \{c[v_i]\}_{i=1\dots|V_\Psi|}$ $such\ that\ g(v_i) := c[v_i]$
29.         if $\mathfrak{M} \vDash_g \Psi$
30.             $result := result \cup \{g\}$
31.         Endif
32.     Hcaerof
33.     return $result$
```

Complexity: $O\big(T(k, m, s)\big) = k * m^2 * s$

7.6 Supplementary algorithms for our pattern detection algorithm

7.6.1 BinaryRelationJoin

- Input: β, π sets of labelled tuples , $Domain, Range$: variables
- Output: a set of labelled tuples

Table 10: Supplementary algorithms - BinaryRelationJoin

```
    $BinaryRelationJoin(\beta, \pi, Domain, Range)$

1.  $result := \{\}$
2.  foreach $c \in \beta$
3.      foreach $r \in \pi$
4.          if $Domain \in L_c \wedge Range \in L_c$ then        if $Domain, Range$ are labels in c
5.              if $c.Domain = r.Domain \wedge c.Range = r.Range$    If the common attributes in $r$ and $c$
                                                               are equal in values, then
6.                  $result := result \cup \{c\}$                   concatenate tuples and return one
                                                               tuple with no duplicated attributes
7.              endif
8.          else if $Domain \in L_c$ then                      if $Domain$ is a label in c
9.              if $c.Domain = r.Domain$                       If the common attributes in $r$ and $c$
                                                               are equal in values, then
10.                 $result := result \cup \{r.Range \cdot c\}$      concatenate tuples and return one
                                                               tuple with no duplicated attributes
11.             endif
12.         else if $Range \in L_c$ then                       if $Range$ is a label in c
13.             if $c.Range = r.Range$                         If the common attributes in $r$ and $c$
                                                               are equal in values, then
14.                 $result := result \cup \{r.Domain \cdot c\}$     concatenate tuples and return one
                                                               tuple with no duplicated attributes
15.             Endif
16.         endif
17.     Hcaerof
18. Hcaerof
19. return $result$
```

7.6.2 UnaryRelationJoin

- Input: β, π sets of labelled tuples , $Domain$ is a variable
- Output: a set of labelled tuples

Table 11: Supplementary algorithms - UnaryRelationJoin

```
   $UnaryRelationJoin(\beta, \pi, Domain)$
1. $result := \{\}$
2. foreach $c \in \beta$
3.         foreach $r \in \pi$
4.                 if $c.Domain = r.Domain$              If the common
5.                         $result := result \cup \{c\}$    attributes in $r$ and $c$
                                                          are equal in values,
                                                          then concatenate
                                                          tuples and return one
                                                          tuple with no
                                                          duplicated attributes
6.                 endif
7.         Hcaerof
8. Hcaerof
9. return $result$
```

7.6.3 Grouping Algorithm

- Input: $\beta, \mathrm{V}_{\Psi}^{0}, \mathrm{V}_{\Psi}^{1}$
- Output: a set of labelled tuples

Table 12: Supplementary algorithms - Grouping Algorithm

```
                    $GroupingAlgorithm(\beta, \mathrm{V}_{\Psi}^{0}, \mathrm{V}_{\Psi}^{1})$

1.  $groupedtuples := \{\}$                                  $groupedtuples$ is HashMap
2.  if $\mathrm{V}_{\Psi}^{1} = \{\}$
3.          return $\beta$
4.  Endif
5.  foreach $c \in \beta$
6.          $key := \{c.v | v \in \mathrm{V}_{\Psi}^{0}\}$
7.          $groupedtuples := groupedtuples \cup \{\langle key, c\rangle\}$
8.  Hcaerof
9.  foreach $c \in \beta$
10.         $key := \{c.v | v \in \mathrm{V}_{\Psi}^{0}\}$
11.         $x = groupedtuples(key)$                         return a tuple in $x$
12.         foreach $v \in V^{1}$
13.                 if $x[v] \neq c[v]$                      not to duplicate
14.                         $x[v] := x[v] \cup \{c[v]\}$
15.                 Endif
16.         Hcaerof
17.         $groupedtuples := groupedtuples \cup \{\langle key, x\rangle\}$
18. hcaerof
19. $result := \{\}$
20. foreach $x \in groupedtuples$                            to get the labelled tuples only
                                                             to be passed as the result
21.         $result := result \cup \{x.getValue\}$
22. Hcaerof
23. return $result$
```

7.7 Comparison analysis

In this section, a theoretical comparison between our pattern detection algorithm (Table 9) and the brute force algorithm (Table 8) will be made. The analysis will start with articulating the important parameters in the comparison analysis. Table 13 shows the analysis and highlights key factors which are used in the analysis.

Parameters:

- $w = |\mathbb{U}_0| + |\mathcal{P}(\mathbb{U}_0)|$, *is the number of 0-dim and 1-dim entities in* $\mathfrak{M}$ where $|\mathcal{P}(\mathbb{U}_0)|$ *is the number of all possible 1-dim entities*
- $n = |\mathrm{V}_\Psi|,\ is\ the\ number\ of\ free\ variables\ (v)\ in\ \Psi,$ $where\ \mathrm{V}_\Psi\ has\ been\ defined\ earlier$
- $s =\ |\Psi|$
- $m = max(|R|)|R \in \mathbb{R}$
- $k = |\mathrm{V}_\Psi^1|$

Table 13: Brute force and our algorithm comparison analysis

Algorithm	**Brute force pattern detection algorithm**	**Efficient pattern detection algorithm**
Complexity	$n * w^n$	$k * m^2 * s$
Explanation	$\Rightarrow \lvert V_\Psi \rvert * (\lvert \mathbb{U}_0 \rvert + \lvert \mathcal{P}(\mathbb{U}_0) \rvert)^{\lvert V_\Psi \rvert}$ $\Rightarrow \lvert V_\Psi \rvert * (\lvert \mathbb{U}_0 \rvert + 2^{\lvert \mathbb{U}_0 \rvert})^{\lvert V_\Psi \rvert}$ $\Rightarrow 2^{\lvert \mathbb{U}_0 \rvert * \lvert V_\Psi \rvert}$ • ***Most likely*** $\alpha = max(\lvert V_\Psi \rvert, \lvert V_\Psi^1 \rvert, \lvert \Psi \rvert) < 12$ • ***Because***: there is no pattern up-to-date which has more than 12 components • ***Therefore***: with ignoring the constant values, algorithm complexity is $\Rightarrow 2^{\lvert \mathbb{U}_0 \rvert}$	$\Rightarrow \lvert V_\Psi^1 \rvert * max(\lvert R \rvert)^2 * \lvert \Psi \rvert$ $\Rightarrow max(\lvert R \rvert)^2$ $R \in \mathbb{R}$ worst case $\lvert V_\Psi^1 \rvert * \lvert \Psi \rvert * \lvert \mathbb{U}_0 \rvert^4$ • as if every entity in $\mathbb{U}_0$ is having the R relation with every entity in $\mathbb{U}_0$ • ***Most likely*** $\alpha = max(\lvert V_\Psi \rvert, \lvert V_\Psi^1 \rvert, \lvert \Psi \rvert) < 12$ • ***Because***: there is no pattern up-to-date which has more than 12 components • ***Therefore***: with ignoring the constant values, the worst case complexity of the proposed algorithm is $\Rightarrow \lvert \mathbb{U}_0 \rvert^4$

P complexity class is a category of algorithms which solve problems in polynomial time. The time complexity function for this class is $O(p(n))$ and it can be bounded. On the other hand, exponential time algorithms are known as the "inefficient" algorithms where the time complexity function for such algorithms cannot be bounded. Any computational problem is regarded as well-solved if there is a polynomial solution for it [122].

NP is an acronym for Non-deterministic Polynomial time. NP denotes a category of algorithms which solve problems in non-deterministic polynomial time. NP includes P and NP-Complete classes. Consequently, NP-Complete class is a subset of NP class and is regarded as the hardest problems in NP. There are a number of examples of NP-Complete class problems, for instance, satisfiability and Hamiltonian path problems [122]. Another related class of complexity is known as NP-Hard class. This class categorises the problems which are as hard as the hardest problems in NP with a possibility that these problems are not in NP [122], [123].

Table 13 shows the results of studying the complexity of both brute-forces pattern detection algorithm (Table 8) and efficient proposed pattern detection algorithm, the brute-force algorithm complexity is $O(2^{|\mathbb{U}_0|})$ and the efficient proposed algorithm (Table 9) complexity in its worst case is $O(|\mathbb{U}_0|^4)$. When saying $n = |\mathbb{U}_0|$where $\mathbb{U}_* \triangleq \mathbb{U}_0 \uplus \mathbb{U}_1 \uplus \dots . \uplus \mathbb{U}_d$ such that $d < 3$, and $\mathfrak{M} = \{\mathbb{U}_*, \mathbb{R}, \mathrm{I}\}$ [38], and $\mathfrak{M}$ is one of the inputs of both algorithms, the complexity of the brute force algorithm can be shown as $O(2^n)$ and the complexity of our proposed efficient algorithm can be shown as $O(n^4)$.

It is important to discuss the possibility when the brute force algorithm can outperform our proposed algorithm. As the complexity of the brute-force algorithm is $O(2^n)$, clearly, it can outperform our proposed algorithm when $n < 16$. However, n represents the number of classes and methods in the source code as $n = |\mathbb{U}_0|$ and $|\mathbb{U}_0|$ is the number of entities of 0-dim in the model $\mathfrak{M}$ of the source code. This means that the brute-force algorithm can outperform our proposed algorithm ONLY when the source code contains less than 16 classes and methods. In addition, when dividing this number as each class in the source code has at least one method, the source code would contain 8 classes with 8 methods for each class. It is obvious that a source code with this number of components can only be regarded as a small or a demo source code.

Medium and large scale source codes are often industrial software and are developed using patterns. These kinds of source codes are hard to understand and are manually checked, unlike source codes with 16 classes and methods. Our proposed algorithm outperforms the brute-force algorithm in the detection of pattern in medium and large scale source codes where pattern detection is a need. Furthermore, from the complexity of our proposed algorithm, it can be seen that it terminates when detecting patterns in medium and large scale source codes, whereas the brute-force algorithm does not.

It is evident that the proposed algorithm (Table 9) is a polynomial algorithm and can be regarded as an efficient solution for the problem of pattern detection in the comparison with the brute-force algorithm (Table 8) which can be classified as NP-Complete algorithm and a far exponential algorithm when it is used in pattern detection in medium or large scale source codes.

Having the above argument, it can be concluded that the proposed algorithm outperforms the brute force algorithm and offers an efficient polynomial solution for the problem of pattern detection when the inputs are a model (representing the program) and a formal visual specification such as Codecharts (representing the pattern which is to be found in the program).

7.8 A dry-run example to compare the Brute force pattern detection algorithm with the proposed efficient algorithm

In this section, a performance comparison between the Brute force pattern detection algorithm (Table 8) and the proposed efficient algorithm (Table 9) will be made and discussed. An arbitrary pattern has been chosen in order to be found in an arbitrary implementation. The pattern is shown in Figure 48. Regarding the implementation, it is a large scale source code which is the source code of the TTP-Toolkit. The model of the source code has extracted the information needed for the purpose of the comparison was. Table 14 shows details needed in the dry-run comparison example:

Table 14: TTP-Toolkit source code key parameters for Dry-run example

$\lvert\mathbb{U}_0\rvert$	2234	number of classes and methods entities
$max(\lvert R\rvert)$	1988	the size of the largest relation set in the $\mathbb{R}$ $which\ is\ \underline{Member}$

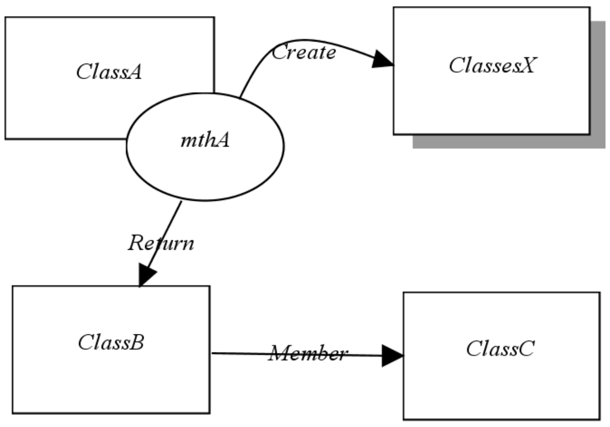


Figure 48: Codecharts: Complexity dry-run example arbitrary pattern

Table 15 shows the dry-run comparison which illustrates how the proposed algorithm significantly outperforms the brute force algorithm (Table 8). Now, it has been proven that the proposed algorithm (Table 9) tackles the problem in a reasonable amount of time. Moreover, as the example of a large scale implementation, the proposed algorithm shows efficiency in such situations.

Table 15: Brute force and our algorithm dry-run example comparison analysis

	Brute force pattern detection algorithm	Proposed efficient pattern detection algorithm
Complexity	$2^{\lvert\mathbb{U}_0\rvert}$	Average case: $max(\lvert R\rvert)^2$ Worst case: $\lvert\mathbb{U}_0\rvert^4$
Result	2^{2234} *=Infinity*	Average case: 1988^2 =15808576 Worst case: 2234^4 = 24907645451536 ***Note: the results are shown in a time unit which costs a computer processor***

Now, it is beneficial to explain how the brute force algorithm (Table 8) and the proposed algorithm (Table 9) work to find the instances of the arbitrary pattern shown in Figure 48. As it can be seen in Figure 48, the pattern has 3 classes namely (*ClassA*, *ClassB*, and *ClassC*). In addition, it has one method named (*mthA*) and a set of classes named (***ClassesX***). In order to find all instances of the pattern, brute force algorithm will, first, generate all the subsets (power sets) of the entities in the source code which are in a set of $\mathbb{U}_0$. Having all the subsets of the 0-dim entities in the model of source code means that the brute force algorithm now have all candidates of the element named (***ClassesX***) in the pattern in Figure 48. As the rest of elements in the patterns are 0-dim elements, their candidates are in $\mathbb{U}_0$. Next, the brute force algorithm will generate all possible combinations of the entities in the source code and the 1-dim entities that were generated early to be candidates for ***ClassesX.*** The combinations will form instances of the pattern. Next, it will create assignments from the pattern to each generated combination. Finally, it will use the TTP Toolkit verifier to check each assignments and save a correct ones.

On the other hand, the proposed algorithm (Table 9) aims to reduce the search space by the means of considering the relations instead of considering the entities. In addition, it expand the 1-dim elements in the patterns and tread them as 0-dim in order to find the correct entities that can be grouped to form a candidate. So, the algorithm will start by retrieving the sets of tuples of relations that are shown in Figure 48 namely *Create*, *Member*, *Call*, and *Return*. Next, it will label the tuples according to the names of elements in the pattern.

For example, let say that the *Return* relation set on the model looks like *Return* ={<java.String.toString(),java.String>,…..,}. The proposed algorithm will retrieve and copy this set as it is one of the relations shown in the pattern, then, the algorithm will label all the tuples in this set according to the elements that have this relation in the pattern. In case of the pattern in Figure 48 the results of this process would look like this *Return* ={<***mthA***:java.String.toString(),***ClassB***:java.String>,…..,}.

After having all the sets of relations copied and labelled, the proposed algorithm (Table 9), will find the instances of the patterns by checking the values according to the labels. By doing so, list of instances will be found and, then, the algorithm will cluster the list according to the entities of labels (mthA, ClassA, ClassB, ClassC). Next, the algorithm will group the entities that has the label ClassesX In order to create the 1-dim entities that can be assigned to ClassesX. Grouping will the ultimate set of entities that can be assigned to the 1-dim element (ClassesX) in the pattern. By doing so, the proposed algorithm (Table 9) will improve on the brute force algorithm (Table 8) by avoiding to generate all the subsets of $\mathbb{U}_0$ to find the sets of entities that can be assigned to ***ClassesX*** to form a correct instance of the pattern. Finally, the proposed algorithm (Table 9) will generate all assignments for the found instances that have been checked in the relations sets and use the TTP Toolkit to verify them.

7.9 Correctness of proposed efficient pattern detection algorithm

In this section, a study on the correctness of our proposed pattern detection algorithm (Table 9) will be carried about and explained. The method used to prove the correctness of the algorithm is the contradiction method. First, the assumptions of the proof will be articulated. Next, some clarifications regarding the algorithm and the proof will be given. Finally, the theorem will be stated and the proof will be shown.

7.9.1 Assumptions

1. ρ is a look-for design or security pattern.
2. $f \overset{\text{def}}{:=} \begin{cases} UnaryRelation(Domain) \\ BinaryRelation(Domain, Range) \\ ALL(UnaryRelation, Domain) \\ TOTAL(BinaryRelation, Domain, Range) \\ ISOMORPHIC(BinaryRelation, Domain, Range) \end{cases}$
3. $\Psi \overset{\text{def}}{:=} \{f\}$
4. Ψ represents ρ
5. $V_\Psi := \{v_1, v_2, \dots, v_k\} | \{v_i\}_{i=1\dots k}$ is free variables in some $f \in \Psi$
6. g_x *is a mapping function denote as* $g_x\colon V_\Psi \to x$
7. $x := \{c_1 \dots\dots c_n\}_{n=1\to|V_\Psi|}$ *where* $c_i \in \mathbb{U}_0$
8. S *represents the set of all instance of* ρ *found by the algorithm*
9. *x is a missed instance of* ρ *iff* $\mathfrak{M} \vDash_{g_x} \Psi$ *and* $x \notin S$

7.9.2 Clarifications

From the above assumptions, the meaning of $\vDash_{g_x} \Psi$ needs to be clarified. It means that $\forall f \in \Psi$, the elements in x , are in a tuple in $R \in \mathbb{R}$, where R **corresponds** to f and these elements correspond to the same variables in all f. When saying R **corresponds** to f, this means that if $f := Inherit(v_1, v_2)$, then the corresponding R is $\underline{Inherit} \in \mathbb{R}$

In order to clarify this more, let's assume that $f := Inherit(v_1, v_2)$ and $f \in \Psi$. Then $\exists c_1, c_2 \in x$ and $\exists \langle c_1, c_2 \rangle \in \underline{Inherit}$ where $\underline{Inherit} \in \mathbb{R}$ and c_1, c_2 replace the same v_1, v_2 in all $f \in \Psi$. For example, lets say $\Psi := Inherit(v_2, v_1) \wedge Member(v_2, v_3) \wedge Aggregate(v_4, v_3)$. Ψ is depicts in the Codechart in Figure 49.

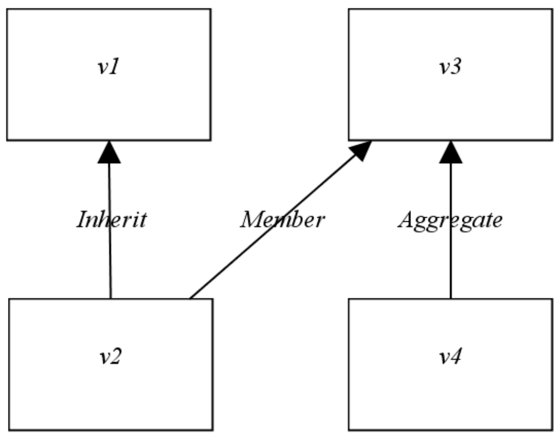


Figure 49: Codechart for algorithm correctness example

Lets assume that $x := \{c_1, c_2, c_3, c_4\}$ and x is an instance of the pattern represented by Ψ. Then x elements need to be in tuples in each time R corresponds to an f ∈ Ψ as follows:

$\langle c_2, c_1 \rangle \in \underline{\mathit{Inherit}}$

$\langle c_2, c_3 \rangle \in \underline{\mathit{Member}}$

$\langle c_4, c_3 \rangle \in \underline{\mathit{Aggregate}}$

And $\{c_1, c_2, c_3, c_4\} \in \mathbb{U}_0$

7.9.3 Theorem

For any Ψ *our algorithm finds all the instances that satisfy* Ψ

7.9.4 Proof of correctness

Assume x is an instance of Ψ and $x \notin$ S. This means that the tuples having the elements of x and belonging to {R} ∈ ℝ (where {R} corresponds to all $f \in \Psi$) are not checked by our algorithm. Thus x is a missed instance of Ψ. Two cases can be identified in order to allow this instance to be missed by our algorithm.

Case 1: a tuple in one of the corresponding R-having elements of x has not been listed for the algorithm to be checked. From the algorithm (Table 9), it can be seen clearly that steps (6, 12) in the main algorithm assure that all tuples in the corresponding R are copied to π and sent to the appropriate ***JoinAlgorithm*** (Table 10or Table 11) for checking**.** Therefore this case cannot occur.

Case 2: a tuple in one of the corresponding R has not been checked. Checking in the ***JoinAlgorithms*** (Table 10and Table 11) is to make sure that an element in x is corresponding to the same variable in all f ∈ Ψ. Again, this case cannot happen as steps (3, 5, 9, and 13) in ***BinaryRelationJoin*** (Table 10) and steps (3 and 4) in ***UnaryRelationJoin*** (Table 11) are to retrieve each tuple in π and check it with the already checked and stored tuples. Furthermore, our algorithm (Table 9) assures that if there are no already checked tuples, all the tuples of the corresponding R (which are listed in π) are returned as already checked tuples. This is made in steps (18, 19, and 20). So, it is not possible to skip any tuple in the checking process.

Finally, form **Cases 1 & 2**, it can be concluded that the assumption of having x as an instance of Ψ and x ∉ S is not a valid assumption and it can be proven the correctness of our algorithm (Table 9).

7.10 Implementation

In this section, a discussion and demonstration of the implementation of our pattern detection algorithm (Table 9) will take place. First, an explanation of the main processes of our pattern detection algorithm will be shown. In addition, the explanation will highlight the integration of the implementation of our approach into the TTP Toolkit. Furthermore, the forms of the inputs and the outputs, which are designed and implemented to interact with the user, will be illustrated. Finally, some of the limitations in the implementation, which are caused by various reasons, will be highlighted.

First, the implementation of the system was developed in the Java programming language. This is due to the fact that the TTP Toolkit was developed using the same language. Therefore, in order to integrate out implementation of our pattern detection approach into the TTP Toolkit, the source code of the TTP Toolkit had to be studied in order to develop the pattern detection implementation in such a way that allows it to integrate into the TTP Toolkit is a part of it.

It is important to illustrate the main processes in the implementation of our pattern detection approach. The implementation relies on the inputs of the user, which are mainly a reversed-engineered program and a pattern in the form of Codecharts. Next, the pattern detection starts and returns a set of potential instances of the patterns. However, these instances need to be verified. In order to verify their design conformance against the given source code, the pattern detection implementation generates an "assignment" which is a form of input that the TTP Toolkit understands and can process.

After the verification of an instance is completed, the result of the verification, then, is tested. If the result is “Passed”, this means that the instance is correct. Next, the implementation will save the “assignment” and will visualise the instance. However, in case that the result is “Failed”, the instance is dismissed. This process will be repeated for all the potential instances. Figure 50 shows the processes explained above.

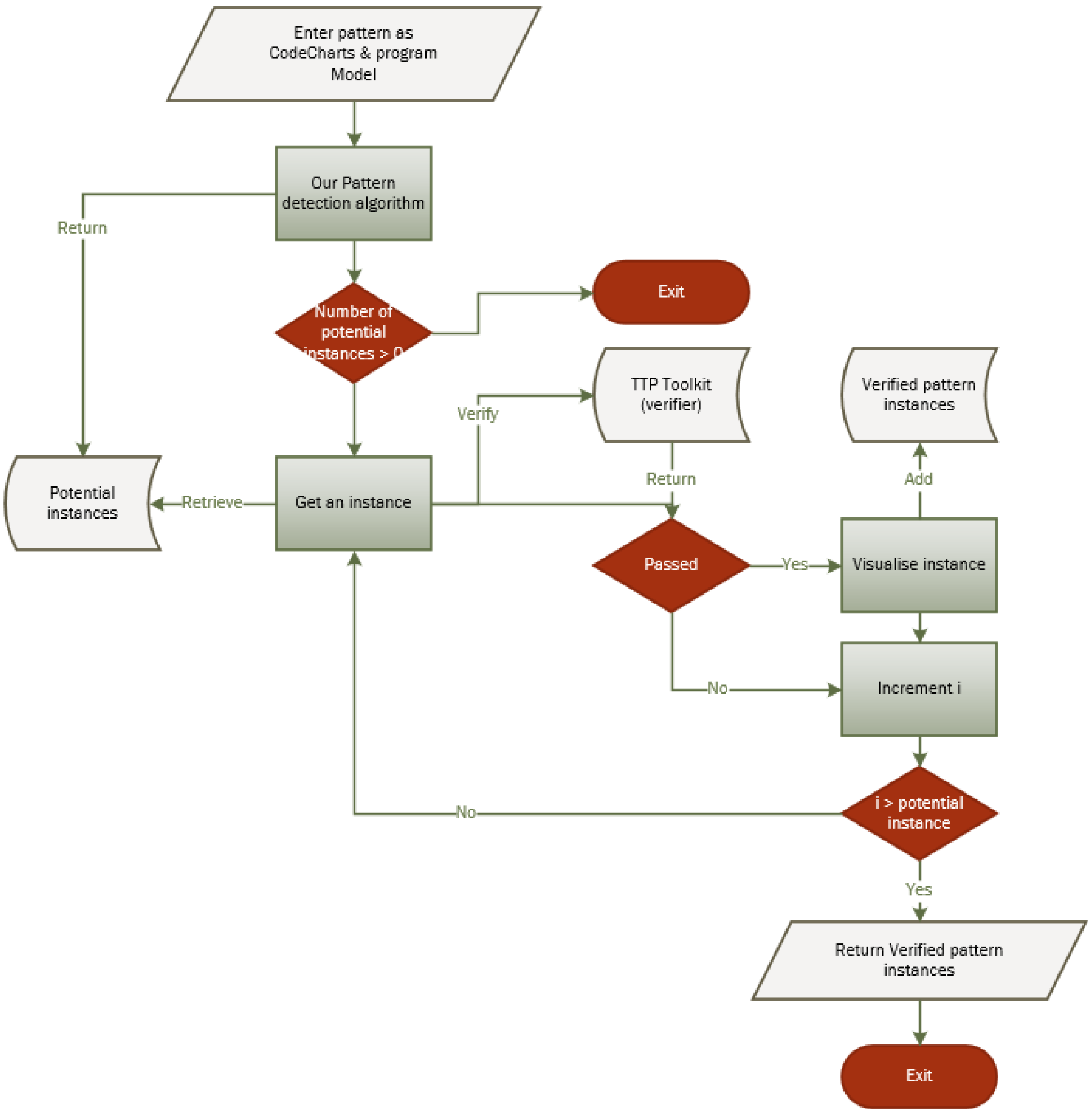
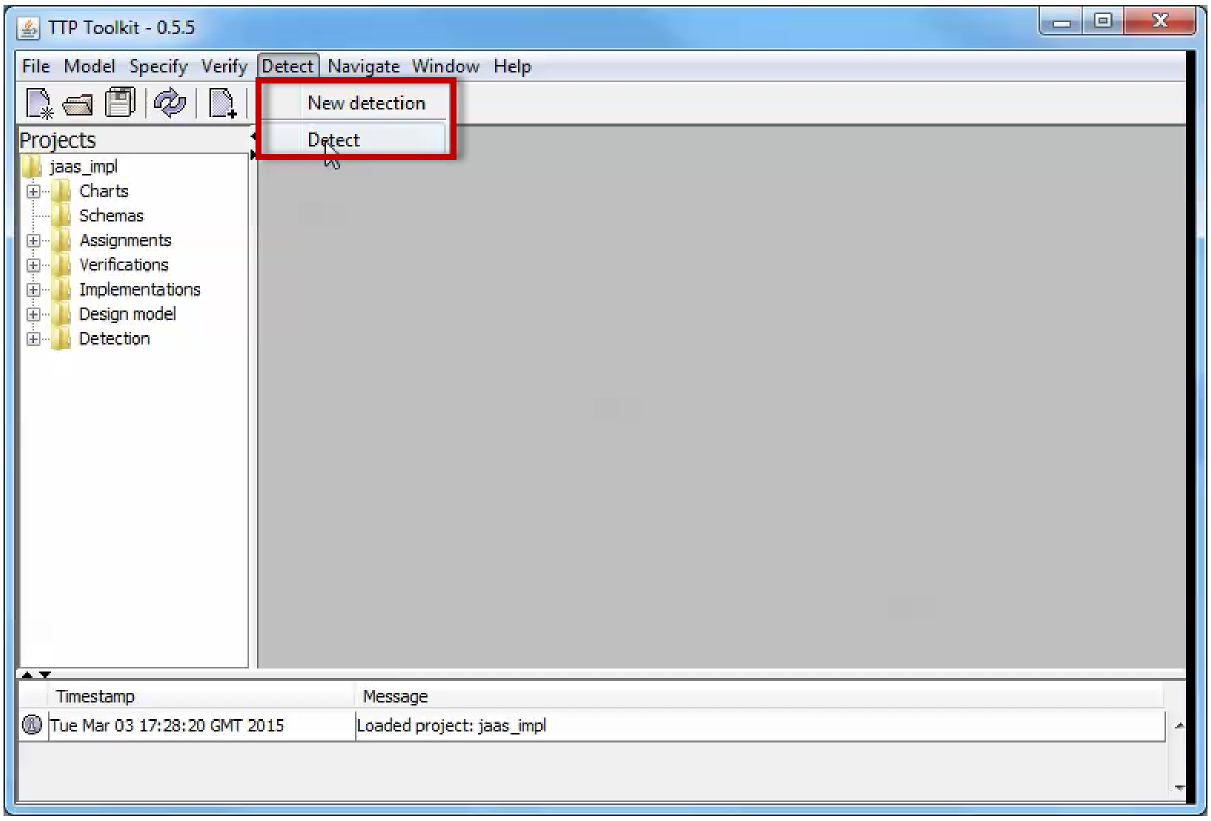

Enter pattern as CodeCharts & program Model
Our Pattern detection algorithm
Return
Number of potential instances > 0
Exit
TTP Toolkit (verifier)
Verified pattern instances
Verify
Return
Add
Potential instances
Retrieve
Get an instance
Passed
Yes
Visualise instance
No
Increment i
No
i > potential instance
Yes
Return Verified pattern instances
Exit

Figure 50: Flowchart for our pattern detection

The main functionality which our implementation has added to the TTP Toolkit is the pattern detection. However, in order to accomplish this main functionality, a number of sub-functions are added as well. These sub-functions are as follows: create new detection entity, and detect (run a detection entity).

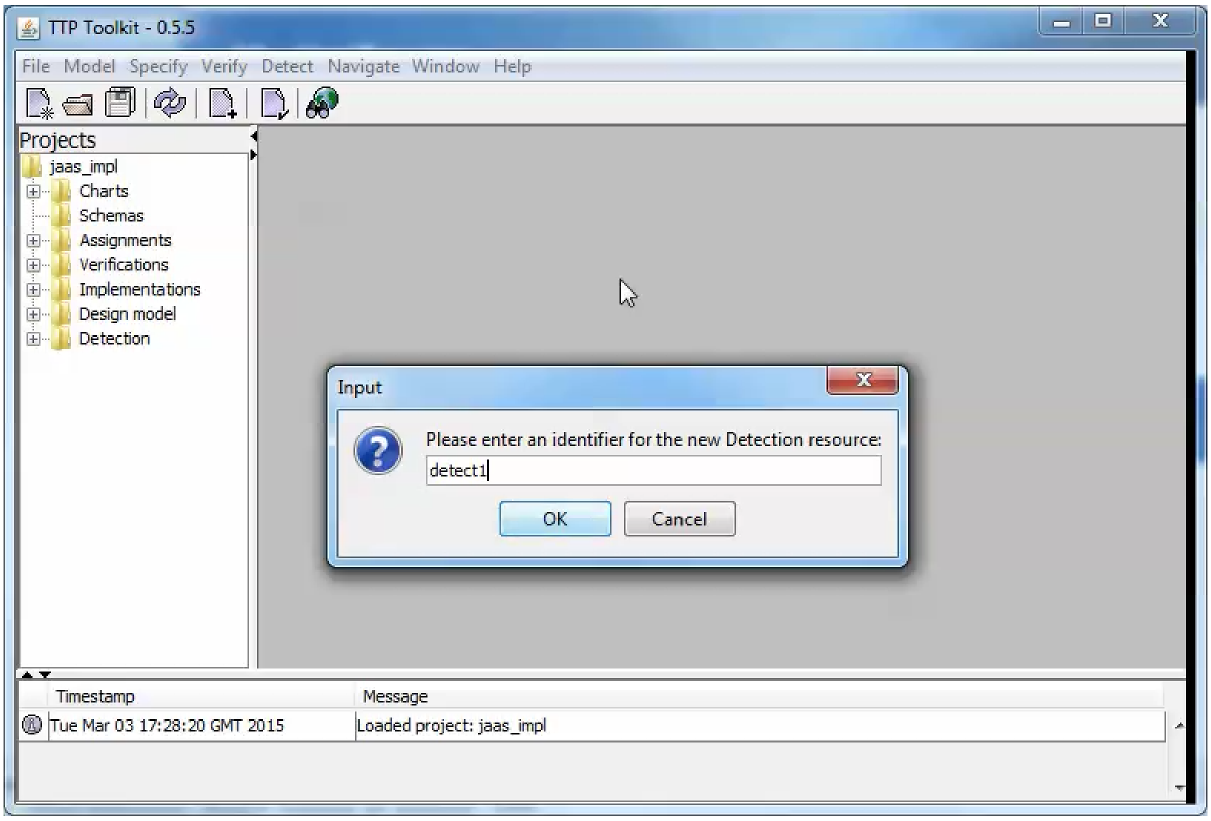
One of the objectives of the implementation of our pattern approach is to integrate it into the TTP Toolkit. In order to achieve this objective, the same conventions (which have been followed in the development of the TTP Toolkit) were adhered to in order to fully integrate our pattern detection implementation into the TTP Toolkit. Therefore, creating a detection entity is a sub-function added for full integration, as the idea of creating entities of process (such as creating chart entity, assignment entity, and verification entity) was in the TTP Toolkit. Figure 51 illustrates the integration of the implementation of our pattern detection approach into the TTP Toolkit Graphical User Interface (GUI). It can be seen clearly from the figure that an item called “detect” has been added into the main menu bar. This item can be expanded to show the main two sub-functions. Furthermore, an item called “detection" has been added in the left hand bar. This item can also be expanded to show the created detection entities, where a user can rename and/or delete a detection entity.



Timestamp	Message
Tue Mar 03 17:28:20 GMT 2015	Loaded project: jaas_impl

Figure 51: Screenshot showing the detection integration in TTP Toolkit

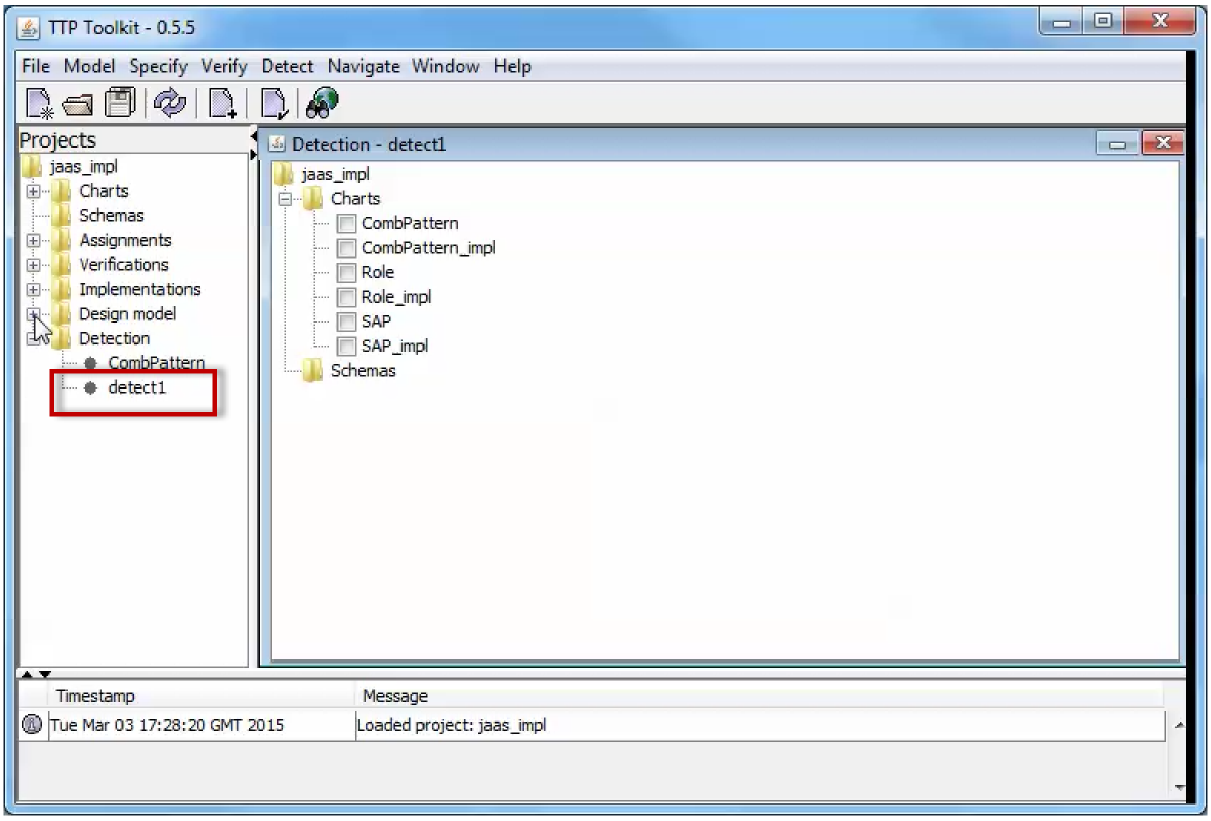
Creating a new detection entity allows a user to detect a pattern. In order to create an entity, first, a user needs to click on the “new detection” in the main menu bar under the “detect” menu’s item. In addition, a user can carry out the same process taking advantage of the drop-down list when mouse-left-clicking on the “detection” item on the left hand bar and going to “New resource”>”Detection”. Either way will open an input window for the user to allow him/her to enter the name for the detection entity. Figure 52 and Figure 53 demonstrate the above mentioned procedures and the outputs of them.



Please enter an identifier for the new Detection resource:
detect1

Timestamp	Message
Tue Mar 03 17:28:20 GMT 2015	Loaded project: jaas_impl

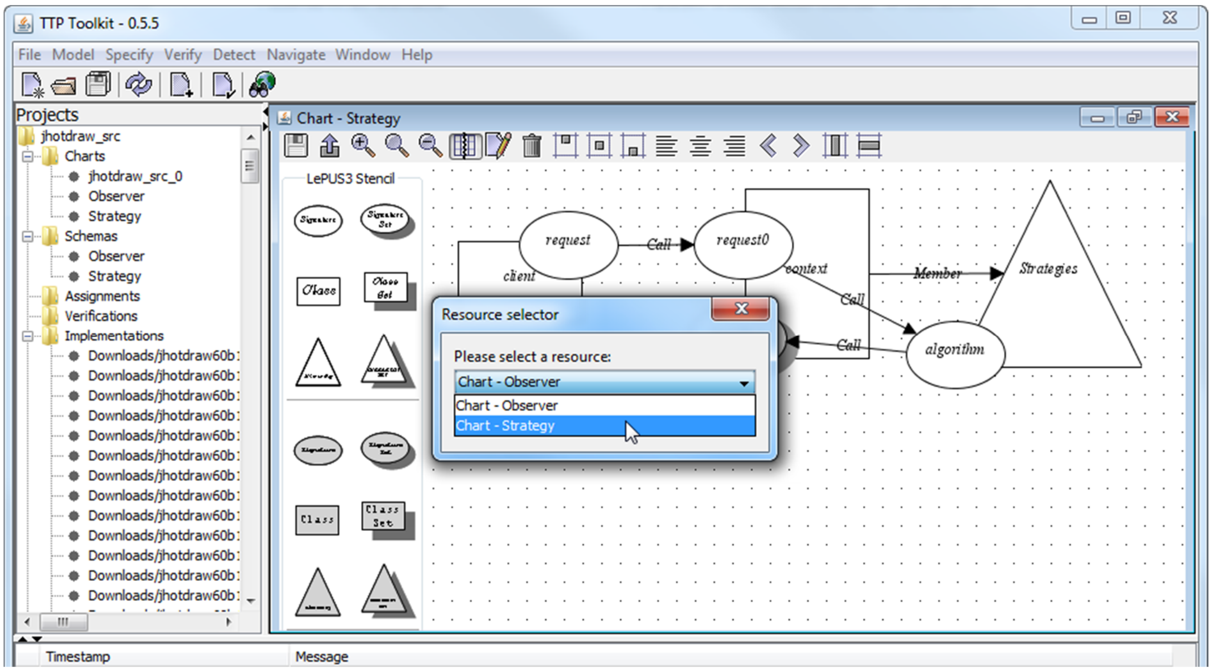
Figure 52: Screenshot showing the detection entity definition in the TTP Toolkit



Timestamp	Message
Tue Mar 03 17:28:20 GMT 2015	Loaded project: jaas_impl

Figure 53: Screenshot showing the detection entity in the TTP Toolkit

After creating the detection entity, what is left is using the second sub-function, which is “detect”. A user of the TTP Toolkit needs to specify the source code of the program that is needed in order to search for pattern instances. Afterwards, the user needs to create a new chart to draw the Codecharts of the pattern into it. Finally, in order to detect the pattern, the user needs to go to the main menu bar of the TTP Toolkit and click on “detect” under the menu of “detect”. This will open a window asking him/her to choose the pattern to be found in the given source code as shown in Figure 54.



Resource selector

Please select a resource:

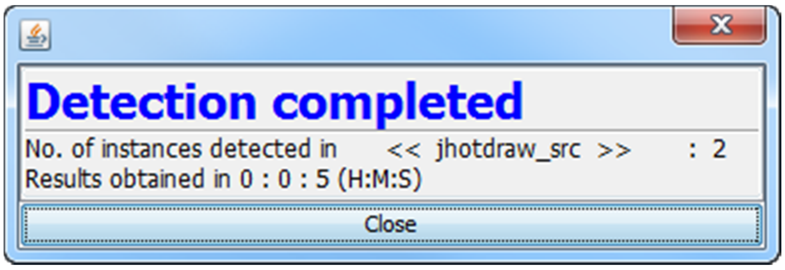
Chart - Observer

Chart - Observer
Chart - Strategy

Figure 54: Screenshot showing the selection of the input pattern for detection

Our pattern detection will start searching for the selected pattern into the source code. This will include the processes described in Figure 49. Immediately after the pattern detection is completed, the main results of the pattern detection process will be summarised and shown to the user as can be seen in Figure 55. The summary shown includes the name of the source code given and the number of detected instances.

Furthermore, the time spent in the entire pattern detection process is shown as well in the summary.



Detection completed

No. of instances detected in << jhotdraw_src >> : 2

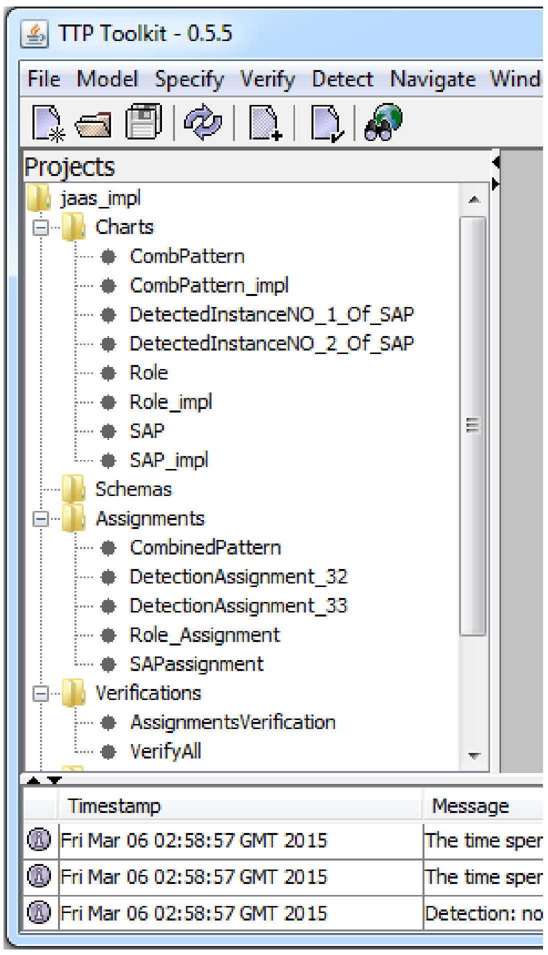
Results obtained in 0 : 0 : 5 (H:M:S)

Figure 55: Screenshot showing the results of detection

Finally, as it has been mentioned earlier in this section, the pattern detection visualises the detected instances. Furthermore, it provides the reasoning for these instances of the available pattern. The reasoning, which is already verified, can be checked by the user at any time. The reasoning is the mappings between the variables in the pattern Codecharts and in the source code components.

The user can, after the detection process is completed, view the visualised pattern instances as well as the mapping (Assignments). Figure 56 shows the left hand bar of the TTP Toolkit GUI where the pattern detection saves the detection results. It can be seen clearly from the figure that the instances of the pattern which have been named according to the following form "DetectedInstanceNO_#_Of_#". This is to distinguish them from other existing Codecharts as well as to allow for indicating their number and pattern.

In addition, Figure 56 shows the mappings which have been used to verify the instances and reason them out. The mappings can be seen in the left hand bar of the TTP Toolkit under the Assignment hierarchy. It can be noticed that the assignment used in the detection is named according to the following form "DetectionAssignment_ #". The number indicates the number of the potential instance of the pattern.



	Timestamp	Message
	Fri Mar 06 02:58:57 GMT 2015	The time sper
	Fri Mar 06 02:58:57 GMT 2015	The time sper
	Fri Mar 06 02:58:57 GMT 2015	Detection: no

Figure 56: Screenshot showing the detection results in the TTP Toolkit

7.11 Summary

In conclusion, in this chapter, the pattern detection problem was considered. This includes clearly articulating the problem definition. Furthermore, the brute force (Table 8) solution was formalised and shown. Besides, our pattern detection solution was discussed and formalised. This includes a description of the way which our solution uses in order to control and minimise the search space in in the source code.

Furthermore, a comparison analysis between the two solutions was carried out and demonstrated. This was alongside with a dry-run example in order to compare the two solutions. Moreover, a proof of the correctness of our pattern detection algorithm was discussed and explained. Finally, the implementation of our pattern detection algorithm was illustrated and explained. This included the illustration of the integration of our algorithm implementation into the TTP Toolkit.

8 Case studies: Pattern Detection

This chapter describes the empirical work of this research in the direction of pattern detection. First, the way in which the case studies are conducted will be described. Second, the open sources used in the case studies will be briefly described. The open sources denote the source codes which have been considered in order to detect the patterns in them.

The rest is divided into two subsections. The first subsection will discuss some case studies carried out in order to detect a number of security patterns formally modelled in Codecharts. The second subsection will discuss Composite pattern case studies in order to show the use of our pattern detection approach to detect design patterns. Finally, a summary of the results and findings of all the case studies will conclude this chapter with tables compiling the results together to be used later in the evaluation chapter of this research.

8.1 Case studies scenario

In this research, case studies are the evaluation methodology. Therefore, it is essential to explain how they are carried out and to clarify all the stages of each case study. The following are the stages which each case study goes through. However, it is important to state here that all the case studies of security patterns rely on the formally modelled security patterns using Codecharts shown in the chapter entitled "Security Patterns Formalisation" and APPENDIX A**.** With regards to the design patterns case studies, the formalised design patterns in the book of Eden et al. [38] were used in design patterns detection case studies.

The first stage, after formally modelling a pattern, is the stage of searching for a claim of pattern implementation in a source code. This stage is basically related to a formalised pattern and source code, which are claimed to be implementing this pattern. The claim is important for the validity of the detected pattern's instances in the source code. Furthermore, it leads to a source code for the case studies. However, some the claims found are for patterns to be implemented in proprietary or commercial source codes [9], [27], which are non-open sources.

The next stage is to find the source code and to extract its model using the TTP Toolkit. Here, the source code needs to be a Java-based code. This is due to the fact that, although the TTP Toolkit is software engineering tools of object-orientation, the current version is only supporting Java-based source codes. After finding the source code which is claimed to implement the looked-for pattern, the TTP Toolkit is used for extracting the model of the source code.

Having the source code model and the Codecharts of the pattern, the next stage is for detecting the instances of the pattern in the extracted model of the source code. This stage comprises using our pattern detection algorithm, auto-generating of the mappings (Assignments) from the patterns components to source code components, auto-using of the TTP Toolkit verifier, and auto-visualising of the correct instances of the pattern.

Finally, showing the results is the last stage. This stage includes showing a summary of the case study. The important information in the summary is the name of the source code used, the number of detected correct instances of the pattern, and the time spent in the detection stage. To sum up, a case study comprises looking for a claim of pattern implementation in a source code, searching for the source code and extracting its model using the TTP Toolkit, detecting the instances of the pattern in the source code, and showing the results.

The pattern detection case studies have been divided into two sets. The first set is gathering the pattern detection case studies on security patterns. The second set is gathering the pattern detection case studies on design pattern. The difference between these two sets of case studies is that the first set relies on the formally modelled security patterns which this research has produced, as shown in the chapter entitled "Security Patterns Formalisation" or in APPENDIX A. The second set of the case studies relies on the already formalised design patterns, which are shown in the book of Eden et al. [38].

8.2 Source codes used in the case studies

The following is a short description of the open source codes which have been considered in the case studies in order to detect patterns in. The models of these source codes have been extracted by the TTP Toolkit [13], [38]. Table 16 shows a statistical summary of the model of each open source, including the numbers of classes and methods as well as the numbers of relations among components (classes and method).

Table 16: Summary of information of source codes used for case studies

Implementation	no. classes	no. methods	no. SignatureOf relation	no. Call relation	no. Forward relation	no. Create relation	no. Return relation	no. Produce relation	no. Member relation	no. Aggregate relation	no. Abstract relation	no. inherit relation
JHotdraw	632	3372	3372	5218	306	1081	198	373	3744	5	423	300
JRefactory	1408	10390	10390	18918	340	4967	459	2136	11542	46	576	773
GanttProject	1040	6941	6941	5811	169	1485	441	318	8374	155	1106	526
Junit	528	1885	1885	661	18	206	12	74	2028	5	46	48
Log4J	338	2298	2298	3162	53	608	153	147	2699	14	107	135
Lexi	138	771	771	852	1	223	39	60	917	3	34	18
JSettlers	179	1725	1725	3983	2	586	195	178	2002	37	10	81
JRISK	137	1287	1287	2697	30	547	80	186	1483	2	44	43

- **JRefactory**

JRefactory is a refactoring tool for Java programing languages. It was developed by Chris Seguin during the period from 1999 to 2002 using Java. Then, the tool was taken over by Mike Atkinson. It is available to download at this link http://jrefactory.sourceforge.net/. JRefactory takes a Java code and generates UML class diagrams. Furthermore, JRefactory allows moving classes between packages, renames fields and other actions, then it makes changes to the given source code accordingly. In addition, it includes a code checker and code metrics. JRefactory supports a number of Integrated Development Environments (IDEs) such as Netbeans, and jEdit [124].

- **JHotdraw**

JHotdraw is a Java framework which aids graphics with drawing and editing. It was mainly developed to exercise design patterns implementation. It is a Graphical User Interface editor. In addition, JHotdraw has been documented at an early stage with regards to design patterns use. Moreover, JHotdraw was designed to be reused and extended. Therefore it can be customised when its structures are understood [125], [126].

JHotdraw is an open source implementation. It is available to download at this link http://www.jhotdraw.org/. Many design pattern detection systems researchers have used JHotdraw to conduce their detection and pattern identification case studies [3], [4], [7], [101]

- **GanttProject**

GanttProject is an open source implementation. It has been developed using Java programing language and it is available at http://www.ganttproject.biz. GanttProject is software for project management and scheduling. It provides many features which help the project administrator in the management. GanttProject is widely used in open source community in order to conduct empirical work as it is considered to be industrial and well-designed implementation [127], [128].

- **JUnit**

JUnit a framework written in/for Java programing language and it is an instance of xUnit architecture for unit testing. It is freely available at http://junit.org/. JUnit provides a facility for test-driven development. It allows for automated testing by providing annotations to methods to be tested as well as many features like assertions, test runners, and others. In addition, it helps with developing more robust software in java [129], [130].

- **Lexi**

Lexi is a free open source word processor. It was developed using Java programming languages. Lexi was introduced by Brill Pappin and Matthew Schmidt in 1999. It has basic features of a word processer. Lexi is freely available at http://lexi.sourceforge.net/. Some design and security patterns are claimed to be implemented in Lexi [3], [9]. Therefore, this open source is used in the case studies of this research.

- **JSettlers**

JSettlers is a java-based game. The game is based on Catan board game [3], [131]. This game's source code has been used in the compared-with pattern detection systems. Therefore, this implementation have been used in order to carry out our case studies on pattern detection. The source code is freely available at http://sourceforge.net/projects/jsettlers/.

- **JRISK**

JRISK is a strategy game that provides support for multi-layers players [3]. The game is originally based on the classical board game. The game has been implemented in Java programing language. A number of versions have been advanced. The source codes and all versions are freely available at http://sourceforge.net/projects/javarisk/files/. In this research, RISK v2.0 has been used in the case studies of pattern detection.

8.3 Detection of security patterns case studies

Using this research pattern detection approach, which has been described in the chapter entitled "Pattern Detection", a number of case studies on security patterns detection in source codes have been conducted. This section reports on some of those case studies. Further case studies can be found in APPENDIX C.

8.3.1 Single Access Point pattern in JAAS.

In the chapter "Case Studies: Formalisation and manual conformance checking of security patterns", a case study on a manual Single Access Point (SAP) pattern conformance checking in JAAS was carried out. The result was finding an instance in JAAS which conform to SAP. However, a number of disadvantages were experienced. One example was the uncertainty of the results that all instances were found. Another disadvantage was the considerable amount of time to manually find an instance and create assignment (mapping from source code components to participants of pattern) in order to verify the detected instance.

Using the pattern detection approach of this research, it was possible to overcome these disadvantages. The SAP pattern, shown in Figure 24, has been detected in JAAS and the results can be seen in Figure 57 and Figure 58. It is clear that the automated detection found 2 instances of SAP in JAAS. The detection process includes finding the instances, generating assignments, using the TTP Toolkit verifier to verify the instances by the generated assignments, and visualising the instances.

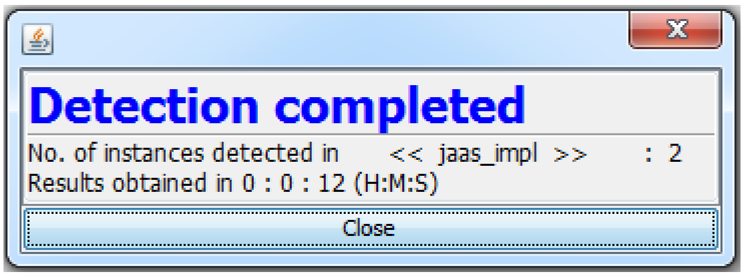
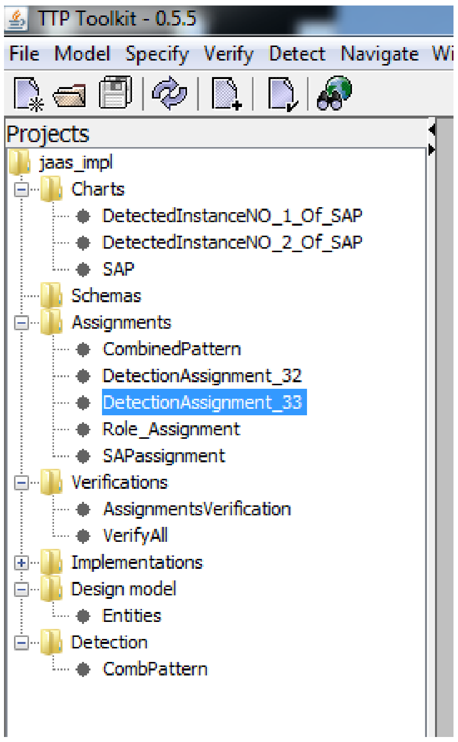

Detection completed
No. of instances detected in << jaas_impl >> : 2
Results obtained in 0 : 0 : 12 (H:M:S)

Figure 57: Results of detecting SAP in JAAS



jaas_impl
Charts
DetectedInstanceNO_1_Of_SAP
DetectedInstanceNO_2_Of_SAP
SAP
Schemas
Assignments
CombinedPattern
DetectionAssignment_32
DetectionAssignment_33
Role_Assignment
SAPassignment
Verifications
AssignmentsVerification
VerifyAll
Implementations
Design model
Entities
Detection
CombPattern

Figure 58: Results of detecting SAP in JAAS shown in the TTP Toolkit sidebar

The 2 detected instances of the SAP pattern has been automatically visualised as can be shown in Figure 59.Figure 60 illustrates the auto-generated assignments of the detected instances to the pattern participants. A number of points need to be clarified here in these instances. The first point is that "LoginContext$4" and "LoginContext$5" are the static objects which the class "LoginContext" Instantiate of itself when "LoginContext" is initialised. Therefore, the TTP Toolkit treats "LoginContext$4" and "LoginContext$5" as independent classes when extracting the model of the JAAS source code.

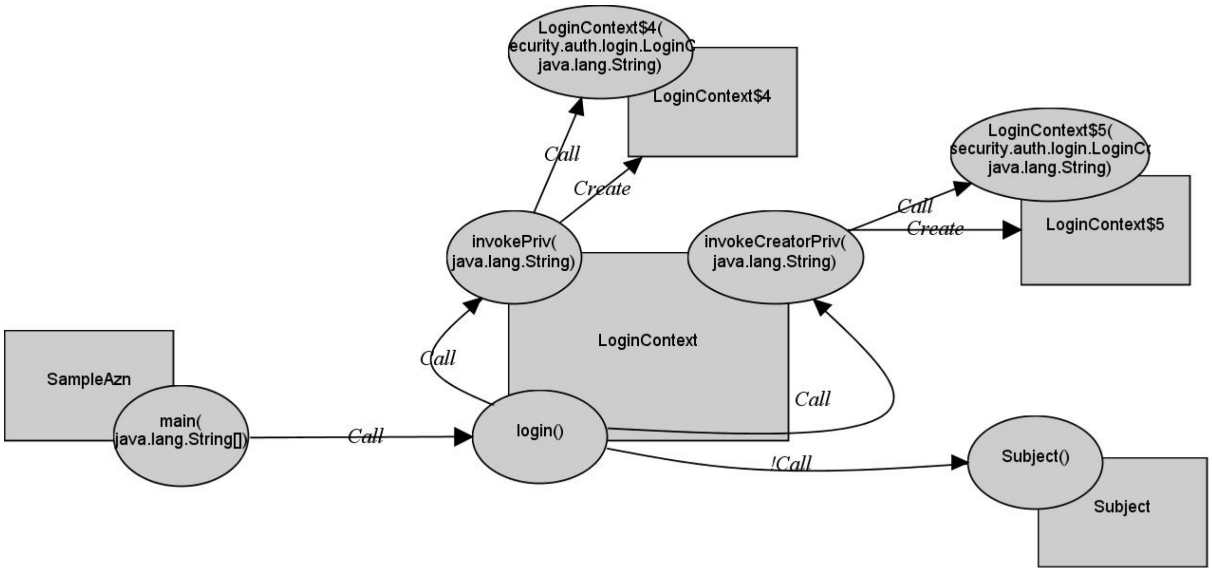
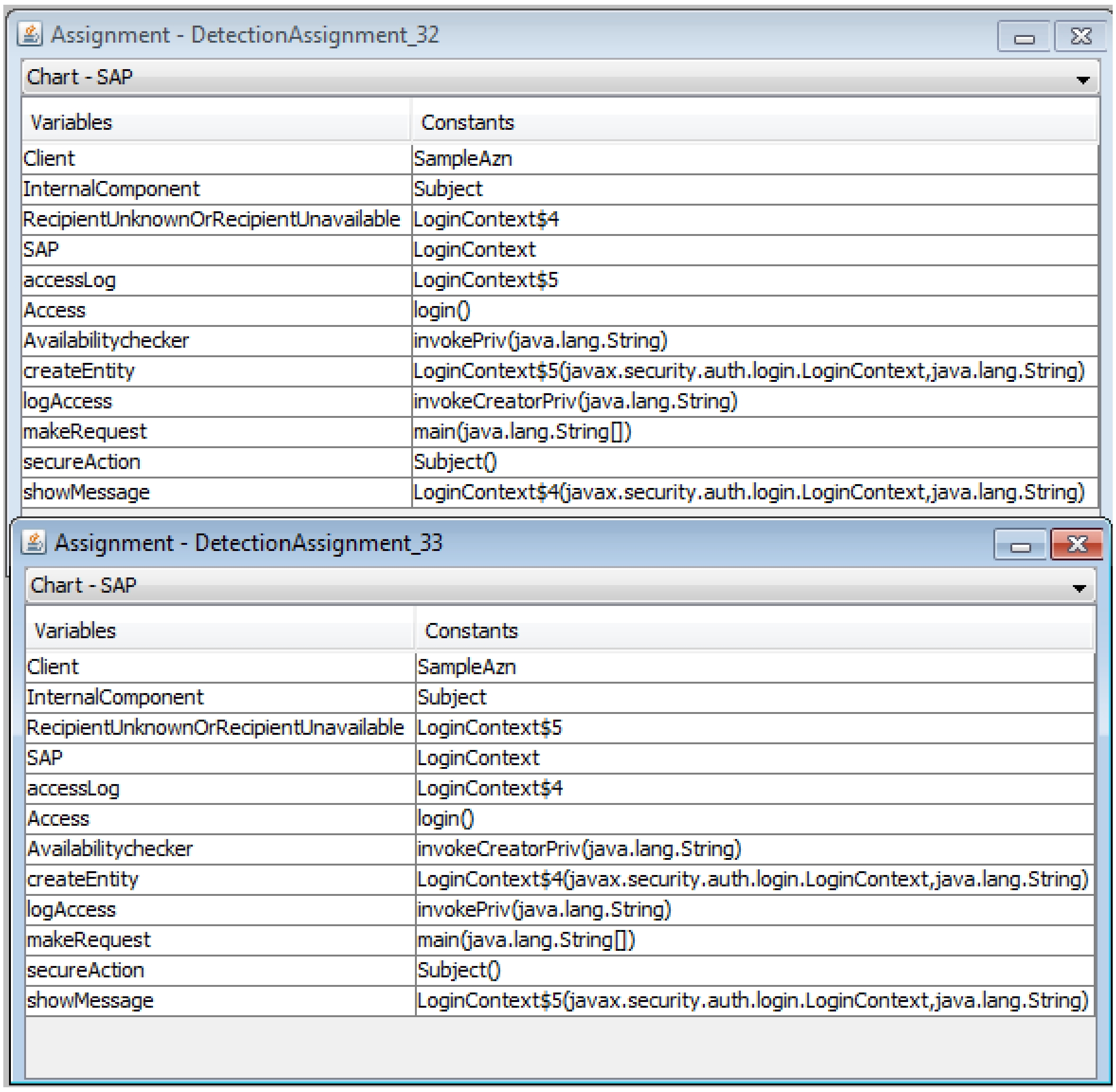


Figure 59: Codechart: An instance of SAP detected in JAAS

Another point is that the "LoginContext$4" and "LoginContext$5" can play both roles of "RecipientUnknownOrRecipientUnavailable" and "accessLog". This is due to the fact that they are configured by a separate text file attached to JAAS. This configuration file is written according to the need. So, the detected instances are similar.



Variables	Constants
Client	SampleAzn
InternalComponent	Subject
RecipientUnknownOrRecipientUnavailable	LoginContext$4
SAP	LoginContext
accessLog	LoginContext$5
Access	login()
Availabilitychecker	invokePriv(java.lang.String)
createEntity	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)
logAccess	invokeCreatorPriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	Subject()
showMessage	LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)

Variables	Constants
Client	SampleAzn
InternalComponent	Subject
RecipientUnknownOrRecipientUnavailable	LoginContext$5
SAP	LoginContext
accessLog	LoginContext$4
Access	login()
Availabilitychecker	invokeCreatorPriv(java.lang.String)
createEntity	LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)
logAccess	invokePriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	Subject()
showMessage	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)

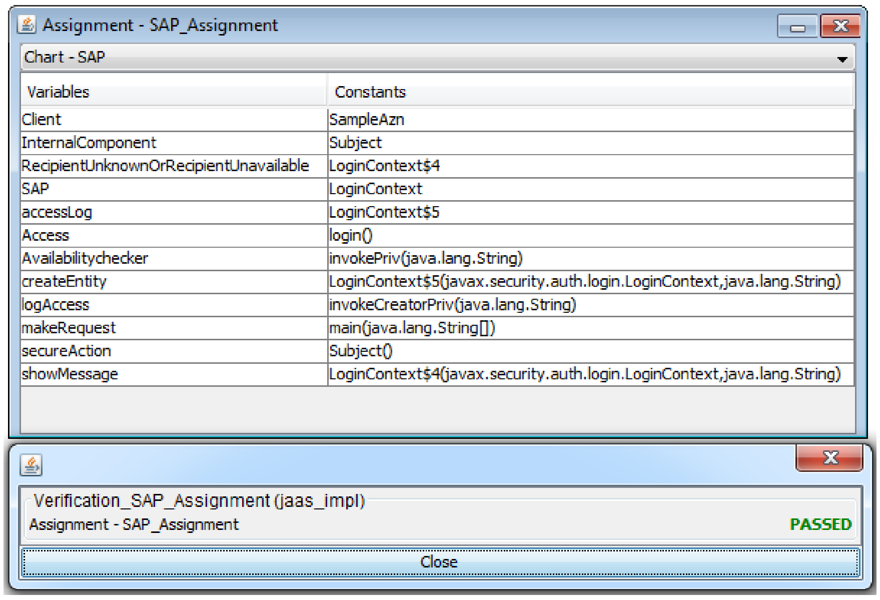
Figure 60: Assignments of SAP detected in JAAS

With a comparison between this case study and the same case study of SAP in JAAS carried out with a manual method and shown in the chapter "Case Studies: Formalisation and manual conformance checking of security patterns" (Chapter 6), a number of points can be highlighted. First of all, the pattern detection approach shows another instance of SAP in JAAS, whereas the manual method showed only one instance of the pattern. Another point is the efficiency in the time of finding, generating assignment, using the TTP Toolkit verifier, and visualising the instances. In the manual SAP checking conformance case study of SAP in JAAS, the time spent was 20 hours;

whereas using the pattern detection approach took only 12 seconds for finding, generating assignments (mappings), verifying by using the TTP Toolkit verifier, and visualising the instances as shown in Figure 57.

Pattern detection aims to find correct instances of a design/security patterns, however, false instances of patterns can occur during the process of pattern detection. So, it is important to articulate what false instances mean. In the context of this research, false instances are the ones that do not conform to the pattern design specification which are drawn in Codecharts.

During manual pattern detection, the source code is manually studied and when a combination of classes and methods are assumed to form an instance of the pattern, an assignment object is manually created and TTP Toolkit verifier is used in order to check whether the instance conforms to the pattern design specification. For example, Figure 61 illustrates an assignment, which has been manually created, and the results of the TTP Toolkit verification on this assignment. As can be seen in the verification results just at the bottom of the figure, the instance is a correct instance and labelled with “PASSED” from the TTP Toolkit verifier.



Assignment - SAP_Assignment

Chart - SAP

Variables	Constants
Client	SampleAzn
InternalComponent	Subject
RecipientUnknownOrRecipientUnavailable	LoginContext$4
SAP	LoginContext
accessLog	LoginContext$5
Access	login()
Availabilitychecker	invokePriv(java.lang.String)
createEntity	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)
logAccess	invokeCreatorPriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	Subject()
showMessage	LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)

Verification_SAP_Assignment (jaas_impl)

Assignment - SAP_Assignment PASSED

Close

Figure 61: correct instance in manual pattern detection process

False instances are checked in the same way in the manual pattern detection. For example, let's take the instance in the previous figure and assume that ALL methods in "*Subject*" class (which represents the *InternalComponent* in the pattern) are secured. Having this assumption we might have a false instance as a result. So, for the purpose of illustration, "*secureAction*" has been assigned to another method named "*isReadOnly()*" which exists in *Subject*". Figure 62 shows that this new combination of components form the source code does not conform to the pattern specification (Codecharts).

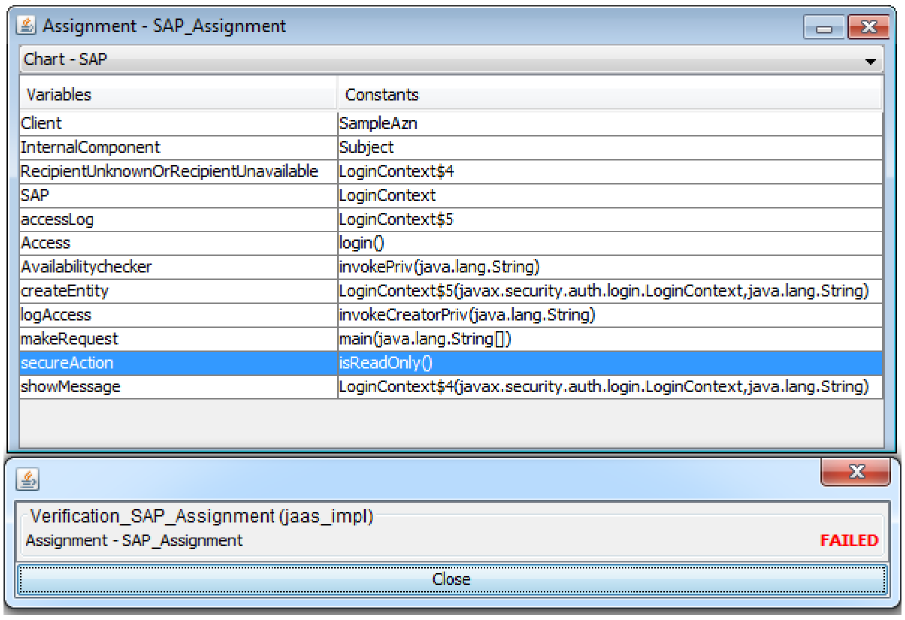

Assignment - SAP_Assignment

Chart - SAP

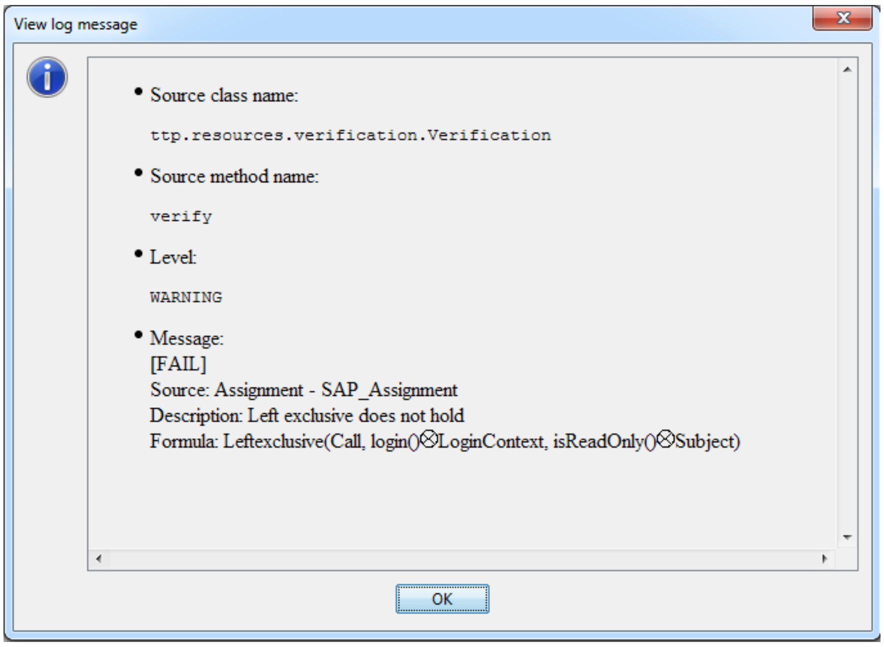
Variables	Constants
Client	SampleAzn
InternalComponent	Subject
RecipientUnknownOrRecipientUnavailable	LoginContext$4
SAP	LoginContext
accessLog	LoginContext$5
Access	login()
Availabilitychecker	invokePriv(java.lang.String)
createEntity	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)
logAccess	invokeCreatorPriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	isReadOnly()
showMessage	LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)

Verification_SAP_Assignment (jaas_impl)

Assignment - SAP_Assignment FAILED

Figure 62: False instance in manual pattern detection process

As can be seen at the bottom of Figure 62, the verification results is “FAILED”, therefore, the instance forms a false instance of the pattern. Moreover, as TTP Toolkit verifier is employed in this research, it provides some reasoning on the false instances. Figure 63 shows the reasoning of regarding the new instance as false instance.



- Source class name:

```
ttp.resources.verification.Verification
```

- Source method name:

```
verify
```

- Level:

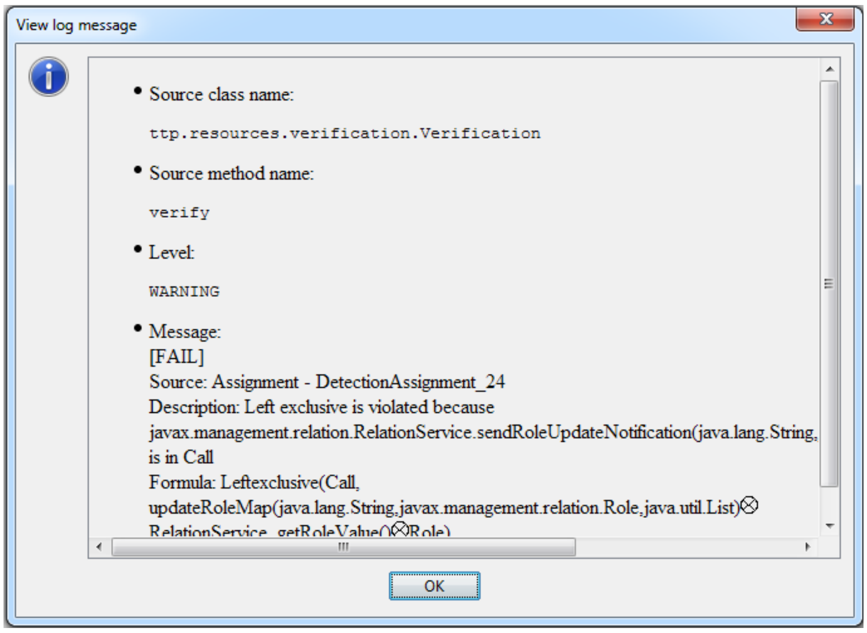
```
WARNING
```

- Message:
[FAIL]
Source: Assignment - SAP_Assignment
Description: Left exclusive does not hold
Formula: Leftexclusive(Call, login()⊗LoginContext, isReadOnly()⊗Subject)

Figure 63: TTP Toolkit reasoning on the false instance – manual detection

In the case of automatic pattern detection, false instances are, firstly, detected while our approach algorithm is checking the source code components that are in relations which the pattern specifies. During the checking, most of the false instances are detected and omitted. Next, our approach employs the same means of the manual pattern detection that has been previously explained. The means are the use of assignments and TTP Toolkit verifier. However, it is important to mention here that our approach will automatically generate assignments for the found instances of the pattern and automatically check those assignments using TTP Toolkit verifier.

In the common case of automatic pattern detection, the false instances are detected during the process of relation checking of the source code components, however, some false instances might occur. In particular, false instances when *Exclusive operator* is used in the specification might occur. So, using the means of assignments and TTP Toolkit verifier will allow detecting false instances and omitting them prior the final detection results is shown. Figure 64 shows the TTP Toolkit verifier reasoning on regarding the instance 24 as false instance, whereas, Figure 65 illustrates the auto-generated assignment named "*DetectionAssignment_24*" which has been created automatically by our approach algorithm and sent to the verifier. As it can be seen from the figures, the instance is regarded as a false instance because of the violation of the exclusive operator. Because of the use of TTP Toolkit, our approach has been able to find that the "*updateRoleMap(String, Role, List)*" is not the ONLY method that *calls* "*getRoleValue()*" method, there is another method named "*sendRoleUpdateNotification(String, Role, List)*" which also calls "*getRoleValue()*" method. According to the specification (Codecharts) of Single Access Point pattern, in Chapter 4 of this thesis, this means that this instance is not conforming to the pattern (in particular specification of exclusive operator) as the "*getRoleValue()*" method (which represents the *secureAction* in the pattern) should be called by ONLY one method in the class that represents the *SAP* in the patterns Codecharts.



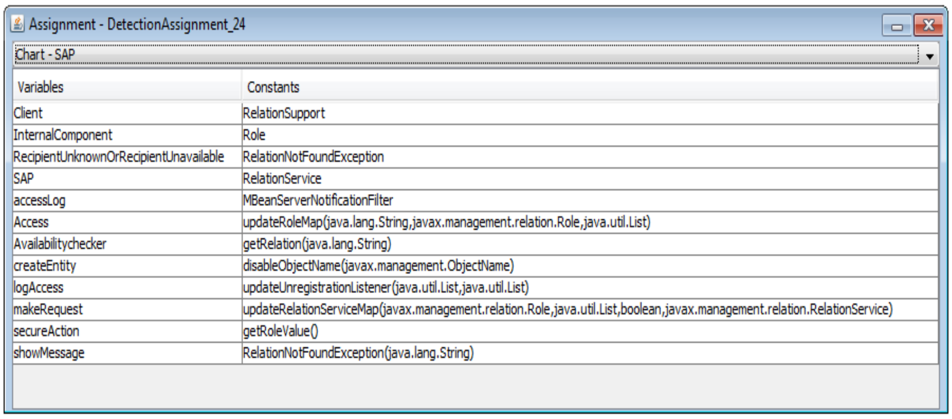
- Source class name:

  `ttp.resources.verification.Verification`
- Source method name:

  `verify`
- Level:

  `WARNING`
- Message:
  [FAIL]
  Source: Assignment - DetectionAssignment_24
  Description: Left exclusive is violated because
  javax.management.relation.RelationService.sendRoleUpdateNotification(java.lang.String,
  is in Call
  Formula: Leftexclusive(Call,
  updateRoleMap(java.lang.String,javax.management.relation.Role,java.util.List)⊗
  RelationService. getRoleValue()⊗Role)

Figure 64: TTP Toolkit reasoning on the false instance – automatic detection



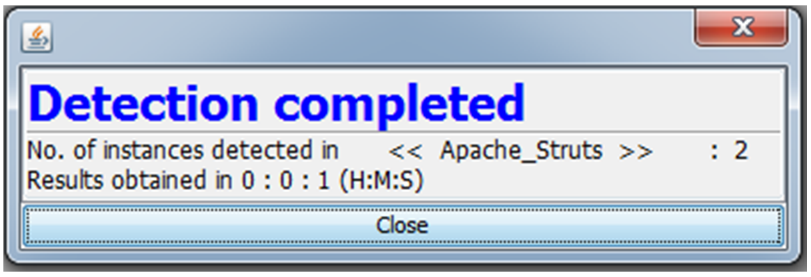
Variables	Constants
Client	RelationSupport
InternalComponent	Role
RecipientUnknownOrRecipientUnavailable	RelationNotFoundException
SAP	RelationService
accessLog	MBeanServerNotificationFilter
Access	updateRoleMap(java.lang.String,javax.management.relation.Role,java.util.List)
Availabilitychecker	getRelation(java.lang.String)
createEntity	disableObjectName(javax.management.ObjectName)
logAccess	updateUnregistrationListener(java.util.List,java.util.List)
makeRequest	updateRelationServiceMap(javax.management.relation.Role,java.util.List,boolean,javax.management.relation.RelationService)
secureAction	getRoleValue()
showMessage	RelationNotFoundException(java.lang.String)

Figure 65: An Example of false instance – automatic detection

8.3.2 Intercepting Validator in Apache Struts

A case study using the pattern detection approach of this research has been carried out to detect the instances of the Intercepting Validator pattern in Apache Struts. The pattern has been discussed and formally modelled in former chapters and shown in Figure 26. This case study demonstrates the pattern detection of a security pattern when 1-dim vocabulary of Codecharts is used in modelling. In the former chapters, the need for 1-dim vocabulary for depicting the abstraction shown in the patterns descriptions has been discussed. In the Intercepting Validator pattern, the 1-dim vocabulary is the hierarchy class variable.

Figure 66 illustrated the outcomes of our pattern detection approach. It is shown that the approach found 2 instances of the Intercepting Validator pattern in the Apache Struts source code. Figure 67 demonstrates one instance which has been automatically checked for conformance and automatically visualised using the approach.



Detection completed

No. of instances detected in << Apache_Struts >> : 2
Results obtained in 0 : 0 : 1 (H:M:S)

Figure 66: Results of detecting the Intercepting Validator in Apache Struts

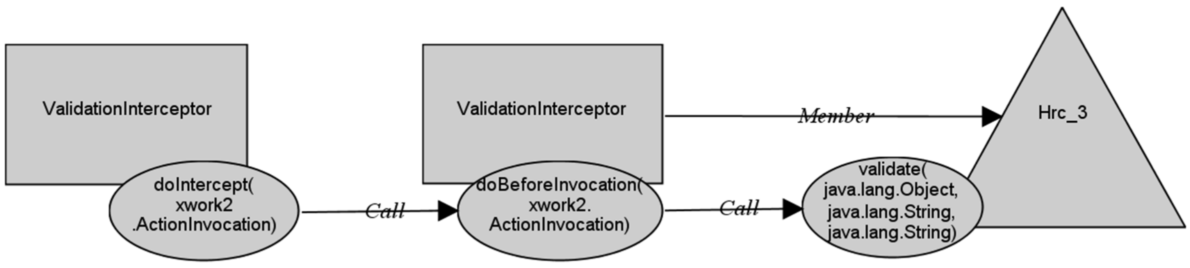

ValidationInterceptor
doIntercept(xwork2 .ActionInvocation)
Call
ValidationInterceptor
doBeforeInvocation(xwork2. ActionInvocation)
Call
Member
validate(java.lang.Object, java.lang.String, java.lang.String)
Hrc_3

Figure 67: Codechart: An instance of the Intercepting Validator detected in Apache Struts

In order to use the TTP Toolkit to check and verify the conformance of the detected instances, our approach automatically generated 2 assignments (mappings) from the Intercepting Validator pattern to the components of the source code, as can be seen in Figure 68. Two points have to be clarified here. The first point is that the component "Hrc_3" is the candidate auto-created by our pattern detection approach to represent the set of classes that play the role of "Validators" participant in the pattern. Figure 69 illustrates the elements of "Hrc_3" and shows the other candidates which have been created by the approach for the other instances which have not passed the conformance checking.

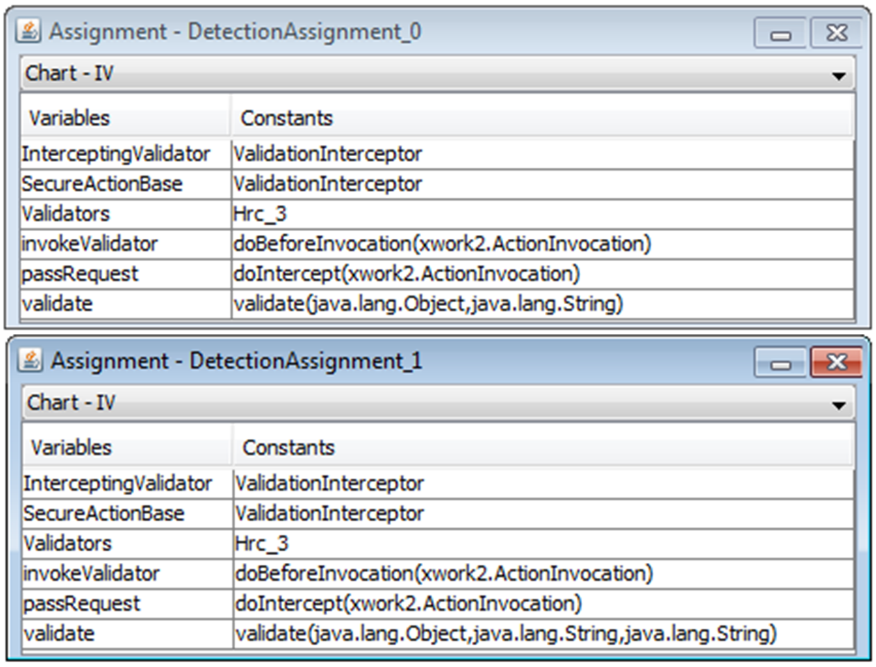

Assignment - DetectionAssignment_0

Chart - IV

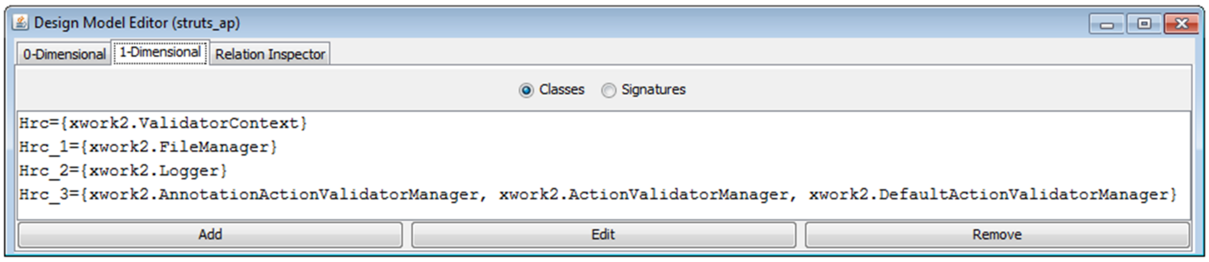
Variables	Constants
InterceptingValidator	ValidationInterceptor
SecureActionBase	ValidationInterceptor
Validators	Hrc_3
invokeValidator	doBeforeInvocation(xwork2.ActionInvocation)
passRequest	doIntercept(xwork2.ActionInvocation)
validate	validate(java.lang.Object,java.lang.String)

Assignment - DetectionAssignment_1

Chart - IV

Variables	Constants
InterceptingValidator	ValidationInterceptor
SecureActionBase	ValidationInterceptor
Validators	Hrc_3
invokeValidator	doBeforeInvocation(xwork2.ActionInvocation)
passRequest	doIntercept(xwork2.ActionInvocation)
validate	validate(java.lang.Object,java.lang.String,java.lang.String)

Figure 68: an assignment of the Intercepting Validator detected in Apache Struts



```
Hrc={xwork2.ValidatorContext}
Hrc_1={xwork2.FileManager}
Hrc_2={xwork2.Logger}
Hrc_3={xwork2.AnnotationActionValidatorManager, xwork2.ActionValidatorManager, xwork2.DefaultActionValidatorManager}
```

Figure 69: Auto-defined higher-dimension entities (Hierarchy) (Intercepting Validator - Apache Struts)

The second point is the similarity between the two detected instances. It can be seen from Figure 68 that the only difference between the instances is the source code components which play the role of the signature variable named “validate”, which is in turn superimposed on the 1-dim hierarchy variable named “Validators”. This is due to the implementation of the pattern in Apache Struts as the method named “doBeforeInvocation(ActionInvocation)” in the class named “ValidationInterceptor” relies on a parameter named “declarative” to determine what validation method (“validate(object, String)” or “validate (object, String, String)”) to call. This parameter should be passed to the “validate” method and the determination process should run there, especially as the “declarative” parameter is of the type Boolean. It is possible to see that the pattern is not implemented correctly in the code; however, this makes the code have two different instances found by our pattern detection approach, which helped to notice deviation in the implementation of the pattern.

In order to invite comparison, the same case study, using the manual method (Chapter 6), took 19 hours, while it took only 1 second with the use of our pattern detection approach . Furthermore, our pattern detection approach found another instance of the pattern, whereas in the manual method, not all instances were found. In addition, all the stages of the manual method of checking conformance have been automated in our pattern detection approach. Moreover, as this pattern has a 1-dim vocabulary variable, it is necessary to create the candidates of this variable. Our approach automatically generates these candidates as shown in Figure 69.

8.4 Detection of design patterns case studies

In order to test our pattern detection approach with other patterns than security pattern and to allow for evaluate it with other similar pattern detection systems that detect design patterns, a number of case studies were conducted to detect a number of well-known design patterns in some open source codes. This section reports on a case study of the detection of the Composite pattern in Java.AWT. However, further case studies can be found in APPENDIX C.

8.4.1 Composite pattern in Java.AWT

Eden et al. [38] have formally modelled the Composite design pattern. Figure 70 shows that pattern according to Eden et al. It can be seen in the figure that a number of different relations are modelled among different types of vocabulary variables of Codecharts. Some of these variables are 1-dim vocabulary and denote abstractions shown in the description of the pattern such as “ComponentOps”, “CompositeOps”, and “Leaves”

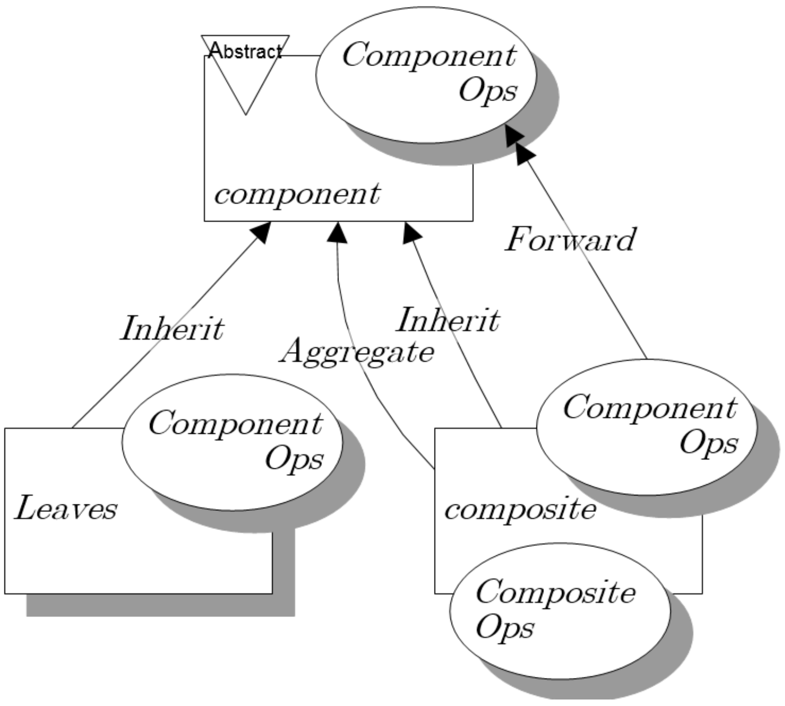
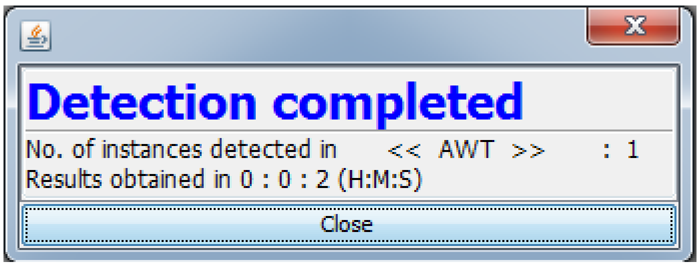


Figure 70: Codechart: Composite pattern [38]

As discussed and explained earlier in the former chapters, our pattern detection approach is introduced to detect any pattern modelled in Codecharts, even when 1-dim vocabulary is used. So, this case study is shown here to allow for expressing the generality of the approach. Figure 71 demonstrates the outcome of the detection process, which took only 2 seconds for detecting the solo instance of the composite pattern. Figure 72 shows the auto-visualisation of the detected instance.



Detection completed

No. of instances detected in << AWT >> : 1
Results obtained in 0 : 0 : 2 (H:M:S)

Figure 71: Results of detecting the Composite pattern in Java.AWT

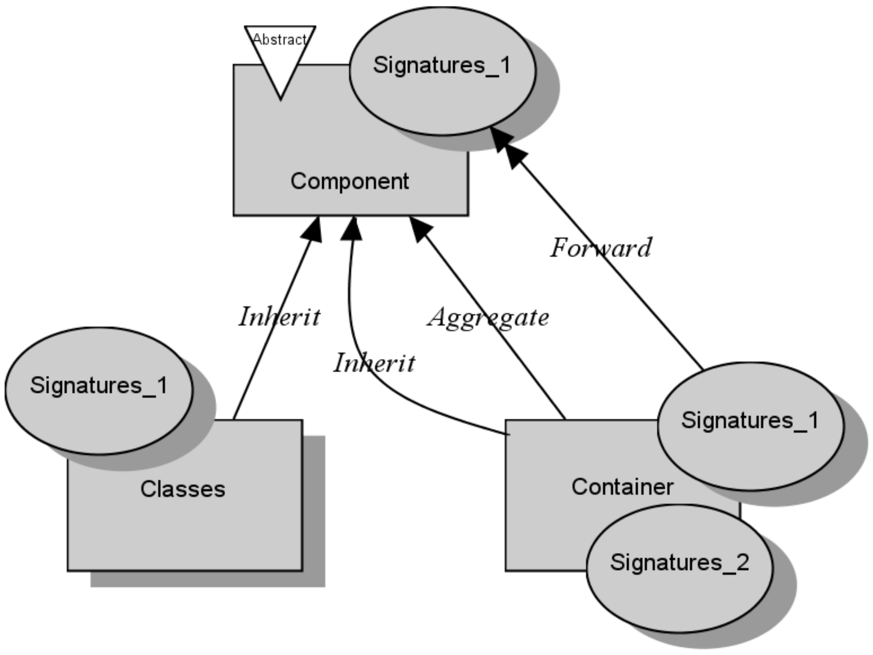
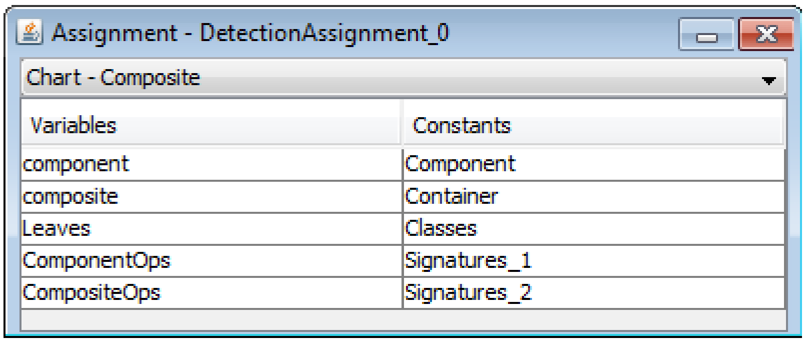


Figure 72: Codechart: An instance of the Composite pattern detected in Java.AWT

The approach has used the TTP Toolkit to verify the instance by the auto-generated assignment shown in Figure 73. As can be seen in the figure, “Classes”, “Signatures_1”, and “Signatures_2” are assigned to the 1-dim variables in the Composite pattern (Figure 70). These are auto-created entities which represent sets of source code components that play the roles of the corresponding participants in the pattern.

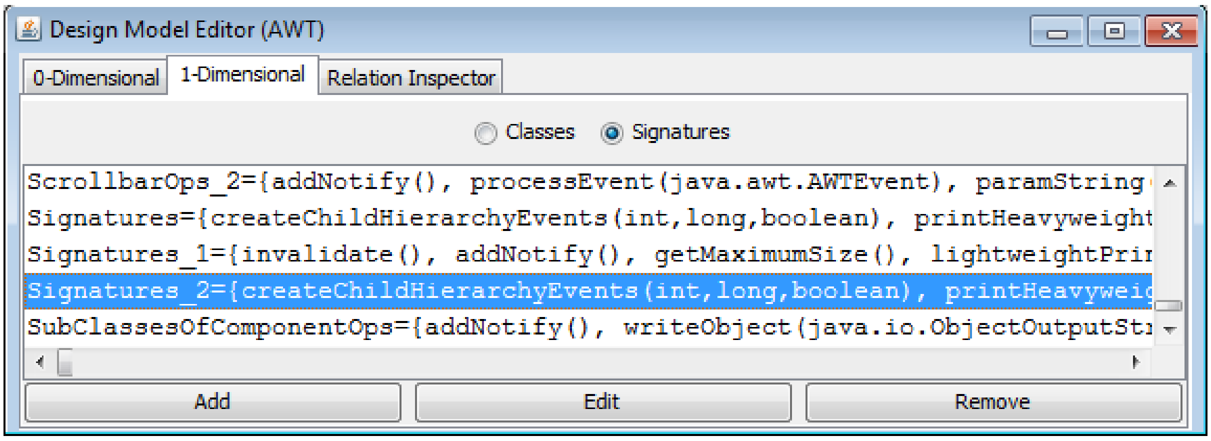


Variables	Constants
component	Component
composite	Container
Leaves	Classes
ComponentOps	Signatures_1
CompositeOps	Signatures_2

Figure 73: An auto-generated assignment of the Composite pattern detected in Java.AWT

Figure 74 illustrates the auto-created 1-dim signature constants (“Signatures_1” and “Signatures_2”) in the TTP Toolkit interface, while Table 17 itemises the actual methods in the source code for each constant. In addition, Figure 75 demonstrates the auto-created 1-dim class constant (“Classes”) in the TTP Toolkit interface.

The auto-created constants are to be assigned to the corresponding variables of the pattern. Each constant abstracts many actual methods in the source code of AWT and plays the role of corresponding variables of the pattern. With this, it can be seen that our pattern detection approach is able to detect any pattern modelled in Codecharts, not only security patterns. In addition, it is able to detect the patterns which are modelled using 1-dim vocabulary that depicts abstraction of the pattern design. Furthermore, it automatically carries all the stages of pattern detection and auto-creates or auto-defines 1-dim constants which are to be assigned to 1-dim variables.



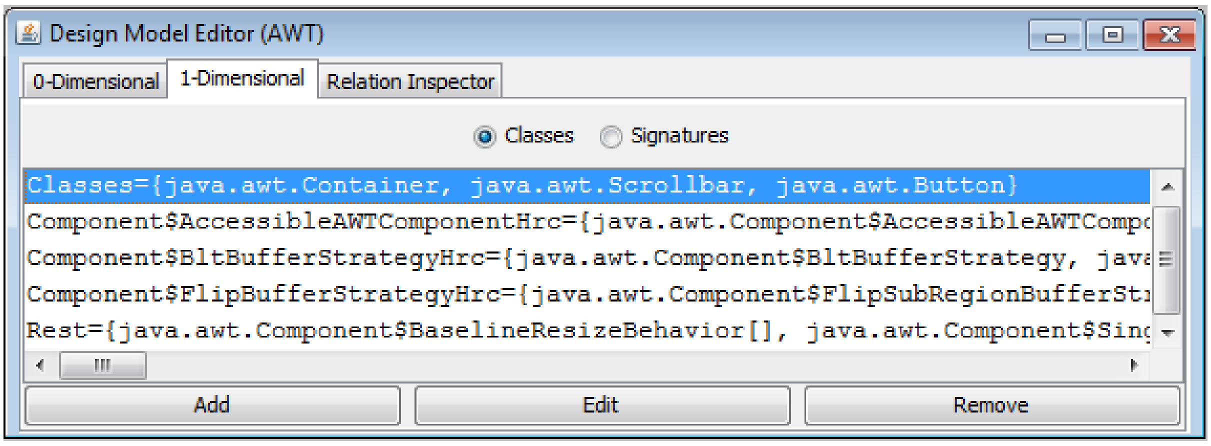
( ) Classes [✓ Signatures]

```
ScrollbarOps_2={addNotify(), processEvent(java.awt.AWTEvent), paramString
Signatures={createChildHierarchyEvents(int,long,boolean), printHeavyweight
Signatures_1={invalidate(), addNotify(), getMaximumSize(), lightweightPri
Signatures_2={createChildHierarchyEvents(int,long,boolean), printHeavywei
SubClassesOfComponentOps={addNotify(), writeObject(java.io.ObjectOutputSt
```

Figure 74: Auto-defined higher-dimension entities (Methods) (Composite pattern - Java.AWT)

Table 17: Details of Signatures sets candidates for Composite instance

Signatures Set	Details
Signatures_1	invalidate(), addNotify(), getMaximumSize(), lightweightPrint(java.awt.Graphics), paramString(), isFocusCycleRoot(java.awt.Container), nextFocusHelper(), checkGD(java.lang.String), addPropertyChangeListener(java.lang.String,java.beans.PropertyChangeListener), applyComponentOrientation(java.awt.ComponentOrientation), preferredSize(), addPropertyChangeListener(java.beans.PropertyChangeListener), print(java.awt.Graphics), eventEnabled(java.awt.AWTEvent), removeNotify(), dispatchEventImpl(java.awt.AWTEvent), lightweightPaint(java.awt.Graphics), list(java.io.PrintWriter,int), transferFocusBackward(), list(java.io.PrintStream,int), minimumSize(), getAlignmentY(), setFont(java.awt.Font), processEvent(java.awt.AWTEvent), getListeners(java.lang.Class), getAlignmentX(), deliverEvent(java.awt.Event)
Signatures_2	createChildHierarchyEvents(int,long,boolean), printHeavyweightComponents(java.awt.Graphics), proxyEnableEvents(long), lightweightPrint(java.awt.Graphics), isFocusCycleRoot(java.awt.Container), getListenersCount(int,boolean), addPropertyChangeListener(java.lang.String,java.beans.PropertyChangeListener), getPreferredSize(), print(java.awt.Graphics), containsFocus(), removeAll(), getContainerListeners(), add(java.awt.Component,java.lang.Object,int), lightweightPaint(java.awt.Graphics), setFocusCycleRoot(boolean), writeObject(java.io.ObjectOutputStream), transferFocusBackward(), clearCurrentFocusCycleRootOnHide(), checkAdding(java.awt.Component,int), remove(java.awt.Component), getAccessibleAt(java.awt.Point), findComponentAt(int,int,boolean), paint(java.awt.Graphics), update(java.awt.Graphics), createHierarchyEvents(int,java.awt.Component,java.awt.Container,long,boolean), add(java.awt.Component,java.lang.Object), getListeners(java.lang.Class), getAlignmentX(), findComponentAt(int,int)



Selected: Classes

```
Classes={java.awt.Container, java.awt.Scrollbar, java.awt.Button}
Component$AccessibleAWTComponentHrc={java.awt.Component$AccessibleAWTCompo
Component$BltBufferStrategyHrc={java.awt.Component$BltBufferStrategy, java
Component$FlipBufferStrategyHrc={java.awt.Component$FlipSubRegionBufferSt
Rest={java.awt.Component$BaselineResizeBehavior[], java.awt.Component$Sinc
```

Figure 75: Auto-defined higher-dimension entities (Classes) (Composite pattern - Java.AWT)

8.5 Summary

In conclusion, the above case studies show the use of our pattern detection algorithm to detect a number of security patterns and design patterns. The detected instances of the patterns have been automatically verified by the TTP Toolkit verifier in order to check their design conformance. In addition, the instances have been automatically visualised using our pattern detection approach.

The verifications for the detected instances of patterns were carried out using the *assignments* (mappings) from a pattern to source code components. The assignments were automatically generated by our pattern detection approach.

In addition, as this research aims to detect abstraction shown in the patterns formalisation in the Codecharts, a number of entities in the detected instances are shown in an abstract manner using Codecharts notations of sets of signatures constant, hierarchy constant, and/or classes constant. For more elaboration on these notations, refer to Figure 5. Those entities are auto-defined using our pattern detection approach.

Table 18 and Table 19 summarise the results of all pattern detection case studies on a number of design and security patterns in different open source codes using our pattern detection approach. In addition, ***NI*** denotes the Number of Instances of a given pattern in a source code, while ***T*** denotes the Time taken to detect, verify, and visualise the detected instances. The time is shown in seconds. The dashes shown in the tables mean that the detection of a pattern on the shown source code was not carried out or did not find correct instances of the pattern. The reason behind the cases of not carrying out the detection is that a claim of an implementation of the pattern in the source code could not be found or the compared-with detection systems have not considered these detection cases.

Table 18: Design Patterns Detection results

Open source / Pattern	JHotdraw		JRefactory		GanttProject		JSettlers		RISK 2.0		Java.AWT	
	NI	T	NI	T	NI	T	NI	T	NI	T	NI	T
Class Adapter	9	120	2	206	8	61	1	11	8	17	1	4
Factory Method	5	46	1	279	-	-	-	-	-	-	-	-
Strategy	1	14	2	145	-	-	-	-	-	-	1	19
Composite	-	-	-	-	-	-	-	-	-	-	1	2

Table 19: Security Patterns Detection results

Open source / Pattern	JAAS		Apache Struts		LEXI	
	NI	T	NI	T	NI	T
SAP	2	12	-	-	-	-
Check Point	8	1	-	-	-	-
SAP & Check Point	12	8	-	-	-	-
Intercepting Validator	-	-	2	1	-	-
Full View with Error	-	-	-	-	2	1

The results above show the outcomes of case studies of pattern detection of 4 design patterns as well as 5 security patterns. The design patterns are Class Adapter, Factory Method, Strategy, and Composite pattern. The security patterns are Single Access Point (SAP), Check Point, SAP & Check Point, Intercepting Validator, and Full View with Error. These patterns were searched for in a number of open source Java-based codes using our pattern detection approach. Finally, the information in the above tables will be used in the evaluation of our pattern detection approach with the other pattern detection systems. Therefore, in the evaluation chapter of this research, these above tables might be referred to for clarity and linking.

9 Results and Evaluation

This chapter focuses on describing the evaluation of the research as well as showing the main findings of the research. The evaluation is divided into two main parts. The first part is to evaluate the research by means of feature analysis with comparison to other state of art pattern detection systems. The second part is to evaluate the system performance.

9.1 Evaluation criteria

Guéhéneuc et al [3] have defined a set of criteria which the current pattern detection systems should meet. The criteria are: high precision, ease-of-use, ability to detect incomplete instances, explanation, and performance efficiency. Fulfilling these criteria is the evaluation method which is followed in order to evaluate this research. However, some other common criteria, which have been sought after in the introductions of other pattern detection systems, are considered as well, such as reasoning, availability, ability to detect new patterns, dynamic checking, and full automation. The following are short descriptions for each criterion.

- ***Good Precision***

Precision can be defined as the percentage of the correct pattern’s instances over all the returned instances. Most design pattern detection systems introduce new approaches to actually improve the precision of the results, which their systems generate in the process of detecting given patterns [4]. High precision is a sought-after feature in any pattern detection system.

- ***Ease-of-use***

It is obvious that pattern detection systems need to be easy to use by the intended users. Usually, users are software engineers, architects, security architects, verifiers, and maintainers. The ease of use is generally measured by the ability of the system to interact easily with the user. In addition, it is measured by means of allowing the users to be able to use the systems’ functionality by the minimum effort. Generally, using the technique of “Click of button” is the best way to measure the system ease of use.

- ***Incomplete instance***

An incomplete instance of a pattern is an instance which represents the occurrence of a subset of the pattern according to the user of detection system preferences. This feature is required by maintainers when a pattern is not determined or is thought to be implemented incompletely. The ability of the pattern detection system to allow detecting incomplete instances of a pattern makes the system more flexible and adaptable.

- ***Explanation***

The pattern detection system should be able to explain the results, which are the detected instances of a given pattern. This explanation includes describing the results, whether they are complete or incomplete instances, as well as describing how the instances are similar to the pattern. Often, visualisation of the detected instances are an effective way of explaining the results.

- ***Performance***

Although pattern detection systems are meant to find instances of a given pattern in a given source code, it has been noticed that some well-known pattern detection systems suffer from serious performance efficiency issues. Therefore, performance efficiency has been a feature of the evaluation of pattern detection systems.

- ***Reasoning***

One of the most important features that any pattern detection system is evaluated with is the ability to reason out the correct as well as the negative pattern's instance. Many pattern detection systems use different reasoners which might be developed especially for that pattern detection system or might be reasoners already available and developed by other parties. However, reasoning the results shows more confidence in the detected instances and allows for theoretical argument of the results.

- ***Availability***

Providing the pattern detection system for academic researchers allows for evaluating the systems by others. This is by making the systems available publicly as well as carrying out the systems evaluation tests on open source implementations, such as JHotdraw [125]. Allowing replications of the evaluation tests on the introduced system is a sign of confidence in and correctness of the introduced system.

- ***Ability to detect new patterns***

Design patterns are evolving, and new patterns are being introduced. It is important for a pattern detection system to be able to detect any new pattern [132]. This feature can be referred to as the "generality" of the pattern detection system. In order to measure the generality of a pattern detection system, two features need to be available in the system. The first feature is that the external inputs (which are the pattern) are not hard-coded in the system but are rather external items which the system can parse and then detect into a given source code. The second feature is the language used to describe the pattern. It is obvious that the language needs to be formal or semi-formal [132]. However, it, in addition, needs to be easy to understand and needs no solid mathematical foundation.

- ***Dynamic checking***

Checking the instances of patterns is carried out either statically or dynamically. The dynamic check involves the execution of a source code and tracing the objects calls and interactions. This check is often conducted after a static check and it is to find or to verify the instances against patterns' behavioural statements.

- ***Full automation***

This criterion measures the ability of the pattern detection system to detect a pattern's instances with no intrusion of human interaction in the searching and following steps until the results are shown. Many systems are regarded as semi-automatic systems, especially those systems which use machine learning techniques. The human interaction might be at minimum; however, it is needed from a person who is experienced in the patterns and in the source code. An interaction of a person with undesirable experience might lead the system to generate incorrect results. This dependency on the human interaction might actually make the system useless when an experienced person is not available. In order to measure whether the pattern detection system is fully automated, the system needs to be able to have the pattern and the source code as the inputs and the results should be generated fully independent from the human interaction.

9.2 Feature analysis comparison

Using the aforementioned criteria, our approach has been evaluated with comparison to a number of state-of-the-art pattern detection systems. Table 20 summarises the features analysis evaluation results. The information in the table was retrieved from the direct use of the available compared-with systems, the original publications of the compared-with systems, and other publications considering the systems [3]–[5], [8], [132]–[134].

Table 20: Features Analysis results

Feature / System	Good precision	Incomplete instance	Availability	ease-of-use	Reasoning	Explanation	Generality	Dynamic checking	fully automated
Our approach	√	√	√	√	√	√	√	×	√
DeMIMA	×	√	√	×	√	√	×	×	×
Similarity Scoring	√	×	×	√	×	×	×	×	×
DP-Miner	√	×	√	√	×	√	×	√	×
PTIDEJ	×	√	√	√	√	√	×	×	×

It can be seen from Table 20 that the competitive approach is the DP-Miner [5] and Similarity Scoring approach [8]. DP-Miner has a feature of dynamic checking of the instances of patterns after static check. Our pattern detection approach is not able to carry out any dynamic check of the patterns instances as Codecharts is limited only to model decidable statements.

However, our approach outperforms the DP-Miner and Similarity Scoring approach by the feature of generality, which allows for detecting a user-specified pattern. This makes our approach general for the detection of any design pattern which is specified in Codecharts, whereas, the other compared-with approaches are limited to a fixed pre-defined set of design patterns.

In addition, our approach has been developed to be full automated, so, no human intervention is required during the detection process, whereas the DP-Miner, Similarity Scoring, and others compared-with approaches required a human intervention at some point in the detection process. This makes them semi-automated pattern detection approaches.

In addition, our approach provides a mean of reasoning on the results of the detection by the use of the TTP Toolkit verifier. This means that it shows that the instances of a given pattern found in the source code are correct instances and have been checked. Moreover, with the use of TTP Toolkit verifier, it was possible to reason on the incorrect instances during the detection process and allow omitting incorrect instances from the final results of the detection.

Similarity Scoring does not have the feature on explaining on the results whether the instances found are complete or not, as it is only find complete instances. In contrast, our pattern detection approach can find complete and incomplete instances of a given pattern and explain on them visually. Incomplete instances can be detected using our pattern detection approach as a user can modify and remove the unwanted elements of a pattern Codecharts then detect the modified version. Moreover, the feature of explanation of the results is provided by our approach as it visualises the instances and the assignments (mappings) from source code components to a given pattern. With regards to the detection of incomplete instances, DP-Miner does not have the ability of detecting them.

Evaluation of the outputs and the results of our approach were conducted by means of case studies. A case study consists of four phases. All the phases are automatically carried out. The first phase (*source code model extraction*) is to have the model of the given source code extracted using the TTP Toolkit. The second phase (*pattern presentation*) is to have a selected pattern visually and formally modelled in Codecharts using the TTP Toolkit. The third phase (*pattern detection*) is to detect and verify the instances of a given pattern against their design conformance to the source code using our approach and the TTP Toolkit verifier. The final phase (*result reporting*) is to visualise all the pattern's instances and the assignments (mappings) from source code components to a given pattern.

In order to carry out the case studies, a number of open source codes have had their models extracted using the TTP Toolkit. In addition, the security patterns, which have been formally modelled in this research, have been used. Regarding the other design patterns, the design patterns shown in [38] have been imported for the purposes of the case studies.

In order to compare the results of our approach with the results of other pattern detection approaches, the same case studies (which have been conducted by other detection approaches) have been replicated. This means detecting same patterns in the same open source codes. So, a number of case studies from the empirical work of this research have been selected. The case studies are to detect four patterns, namely ***Adapter***, ***Factory Method***, ***Strategy,*** and ***Composite*** patterns, in three different source codes named JHotdraw, JRefactory and Java.AWT. The source codes are industrial large scale source codes, i.e. they are regarded as good case studies inputs.

One of the main aims of our approach was to reach the highest precision in the results of the detection. Figure 76 illustrates the precision comparison of our approach with other pattern detection systems. Equation 1 illustrates the common way of calculating the precision of the results of pattern detection systems, where ***TP*** denotes the number of true positive instances and ***FP*** denotes the number of false positives instances.

Equation 1: Pattern Detection Systems Precision

$$P := \frac{TP}{(TP + FP)}$$

The precision ratio is affected by the number of false positives. If a detection system detects a high number of instances, generally, most of them are false positives. Pattern detection systems vary in the way of removing those false positives. Some systems remove them manually; others remove them automatically. However, false positives occur commonly in systems that rely on machine learning, graph matching, and/or information retrieval algorithms.

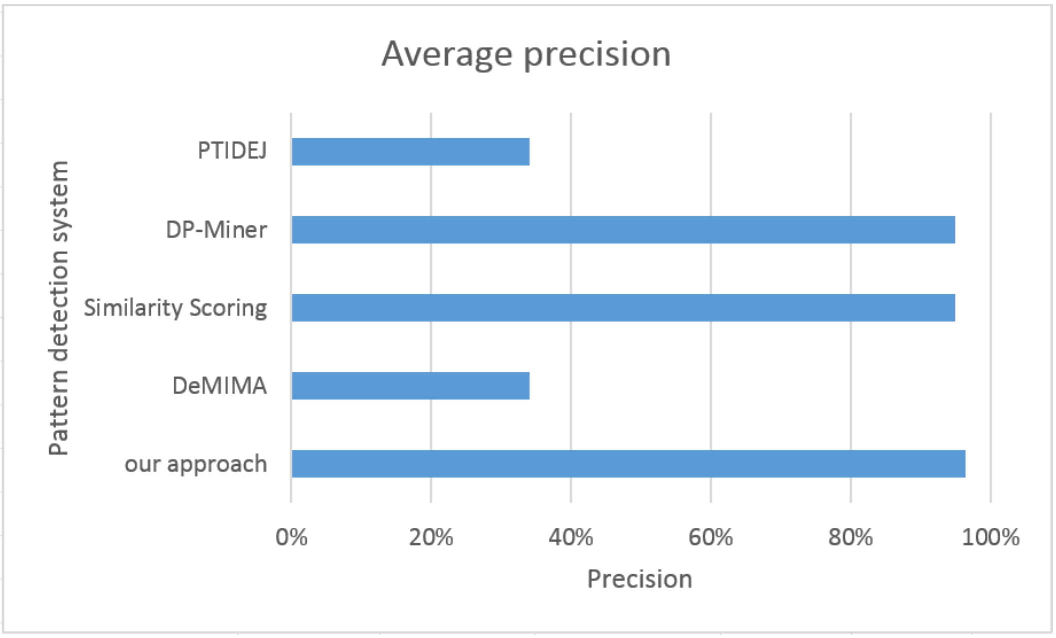


Figure 76: Average Precision of pattern detection systems

With comparison to two state-of-the-art pattern detection systems with highest precision, it can be seen in Figure 77, Figure 78, and Figure 79 that our approach produces the lowest number of instances of the given patterns. This is due to two reasons. The first reason is that the patterns are formally described with rich structural information, as our approach uses Codecharts, unlike other systems which rely on UML diagrams and/or information retrieval parameters such as names and comments of source code components, which might be misleading in most cases. Furthermore, Codecharts can capture the abstraction in the patterns with the use of 1-dim notations as well as hierarchy notations. This allows our approach to minimise the number of patterns by encapsulating many instances that differ in one of the pattern's participants, which is meant to be modelled as an abstract set. The second reason is that our approach, as part of the detection process, relies on a verifier which takes the instances and verifies them against the source code.

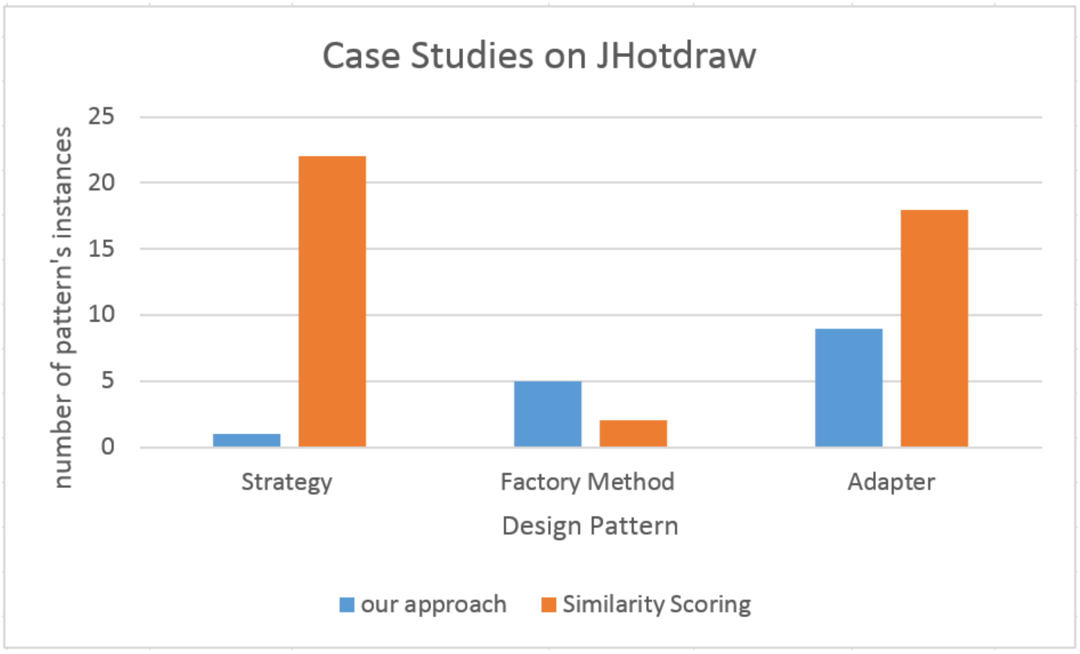


Figure 77: Instances numbers of JHotdraw case studies compared with the Similarity Scoring system

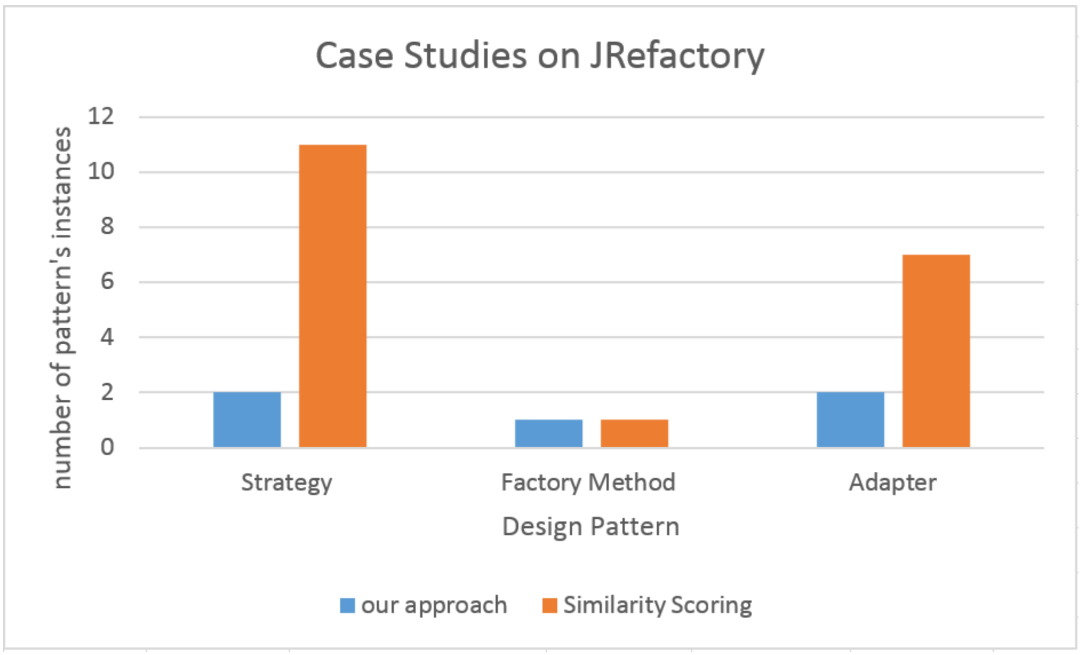


Figure 78: Instances numbers of JRefactory case studies compared with the Similarity Scoring system

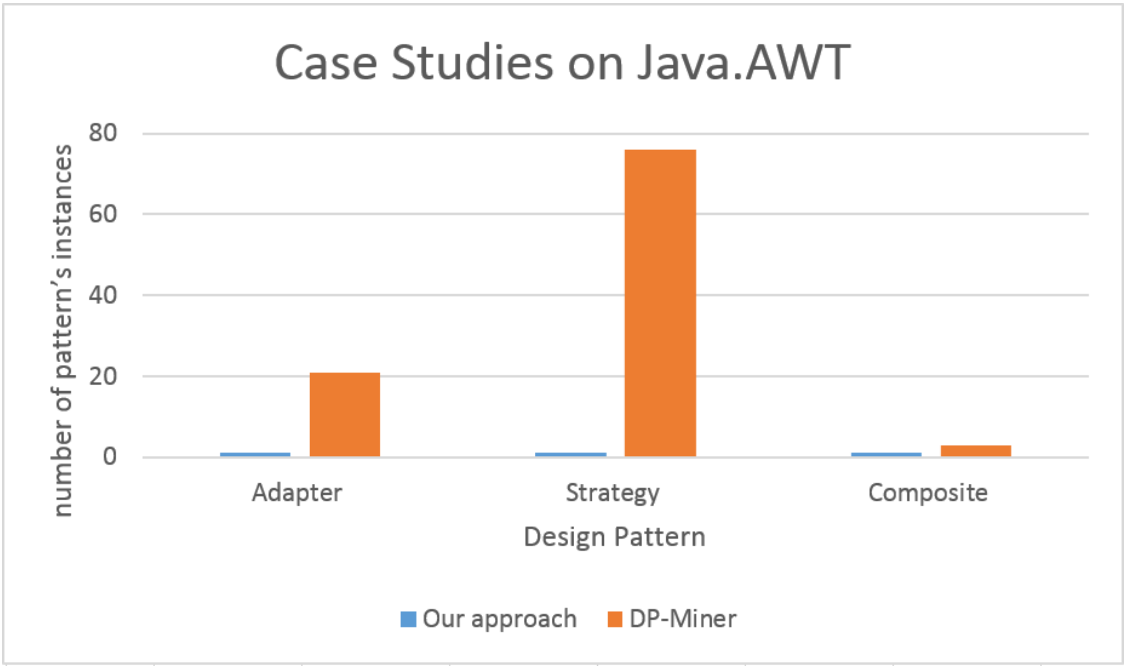


Figure 79: Instances numbers of Java.AWT case studies compared with the DP-Miner system

The Similarity scoring approach [8] is currently the pattern detection approach with the highest precision ratio. Having this in mind, it is, in addition, the approach with the lowest number of instances for the given patterns in the given source codes. It can be seen clearly from Figure 77 and Figure 78 that our approach reaches the closest number of instances in the same case studies.

Moreover, our approach has a better lower number of instances in the case study results shown in Figure 77 and Figure 78. This is due the detection of abstraction, which the scoring similarity does not consider. The abstraction is in the strategy role of the Strategy pattern. The similarity scoring represents it as three classes or perhaps more, whereas, this research approach detect it using Codecharts notations which represent it as one hierarchy.

The lower number of instances of patterns, illustrated in Figure 77, Figure 78, and Figure 79, shows that our pattern detection approach does not return many false positive instances, which improves the overall precision of the results. In addition, it means that the instances of the patterns in the source code are found with the original abstraction level of encapsulation and information hiding, which is described in the patterns' original catalogues. This is due to the use of 1-dim notations of Codecharts (hierarchy, 1-dim signature variable, and 1-dim class variable), unlike the other compared-with pattern detection systems which use UML diagram to model the patterns.

9.3 Evaluation performance

Efficiency is one of the most important features which any pattern detection system aims to acquire. Efficiency, here, refers to the overall performance of pattern detection approaches. This section describes the efficiency of our approach in detecting patterns in different source codes with comparison to other pattern detection systems.

In order to evaluate the performance of our approach, the time spent for each case study has been automatically calculated. This was by the means of capturing the time before the detection starts and capturing the time after the results of detection are verified and visualised. The calculated time is presented for each case study as shown in the previous chapter.

Next, the case studies, which are the same to the ones that other pattern detection systems have conducted, have been replicated. This means selecting, from this research empirical works, the case studies for the patterns and the source code which other pattern detection systems have considered. So, the Adapter pattern case studies, in two different source codes, namely JHotdraw (4004 components) and JRefactory (11798 components), have been selected. In addition, The Factory Method and Strategy pattern case studies, on the same aforementioned source codes, have been selected. Those case studies have been conducted by two pattern detection systems, whose results are available and published.

The performance has been measured with two parameters. The first parameter is the time taken to carry out the case study. The second parameter is the source code size. There are different ways to measure the source code size. Many measure it with the number of code lines. Others rely on the number of classes and methods in the source code. However, for object-oriented source codes, the number of classes and methods is widely used in order to say how large a source code is. Therefore, and as this research aims to detect patterns from object-oriented source codes, it has been decided to consider the number of classes and methods as the measurement means for the size of source codes. Therefore, when stating the number of source code components, it refers to the number of methods and classes in this source code.

Figure 80, Figure 81, and Figure 82 show the performance of our approach in comparison with the DeMIMA [4] and Similarity Scoring system [8]. To the best of our knowledge, up-to-date, the Similarity Scoring system is the most efficient pattern detection system [3]. It can be seen clearly from the figures that our approach outperforms the DeMIMA system. Furthermore, our approach is competitive to the Similarity Scoring approach, which is the current most efficient pattern detection system.

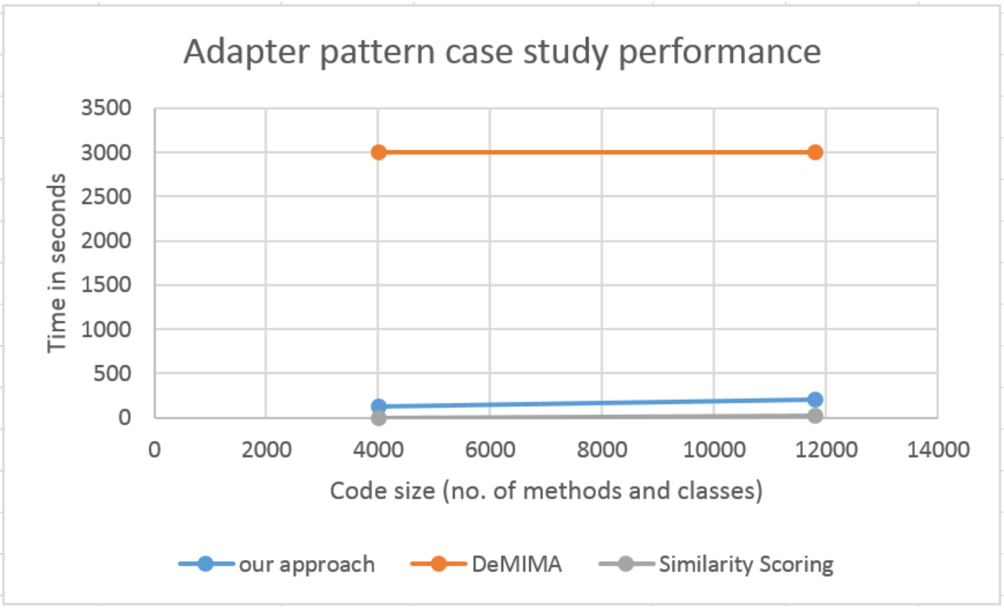


Figure 80: Adapter pattern case study performance

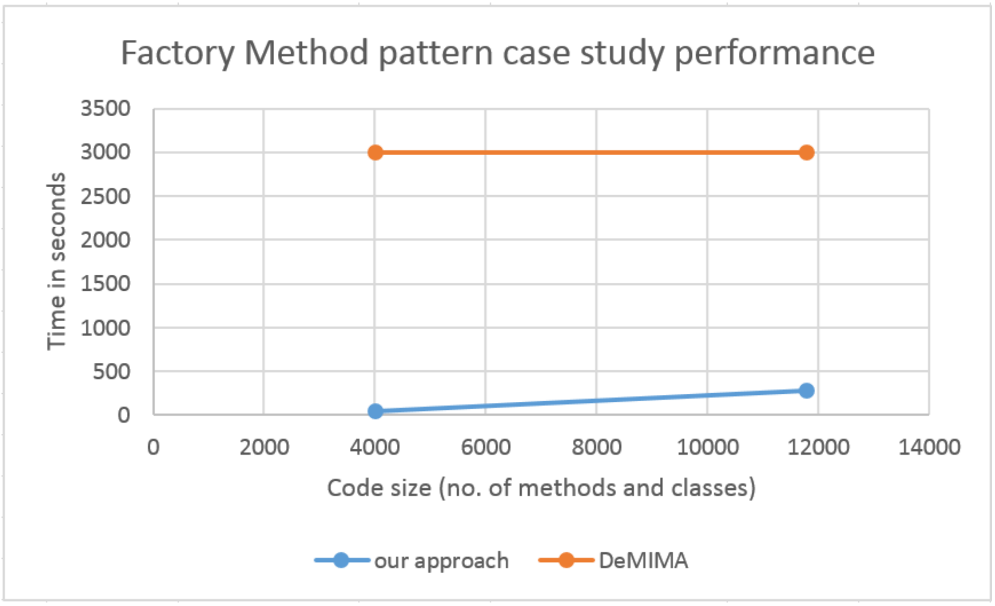


Figure 81: Factory Method pattern case study performance

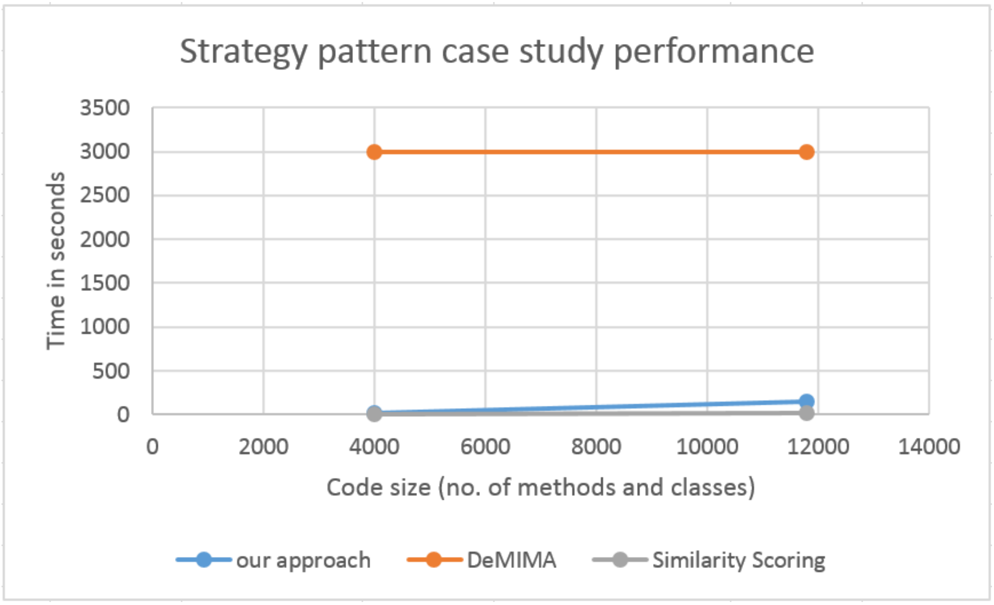


Figure 82: Strategy pattern case study performance

Comparing our approach with more pattern detection systems is a scientific way to evaluate our approach effectively and correctly. Therefore, the comparison of the performance of our approach with others was not limited to the DeMIMA and Similarity Scoring systems. Furthermore, our approach performance was compared with another pattern detection system called Ptidej.

So, from this research set of case studies, three case studies of detecting the Adapter pattern have been selected which were conducted to detect Adapter pattern in three different source codes, namely JSettlers (1904 components), GanttProject (7981 components), and RISK (1424components). Figure 83 shows the comparison between our approach and Ptidej in terms of detecting the Adapter pattern in the same aforementioned source codes. It can be seen in the figure that the x-axis represents the source code size and the y-axis represents the time taken in seconds.

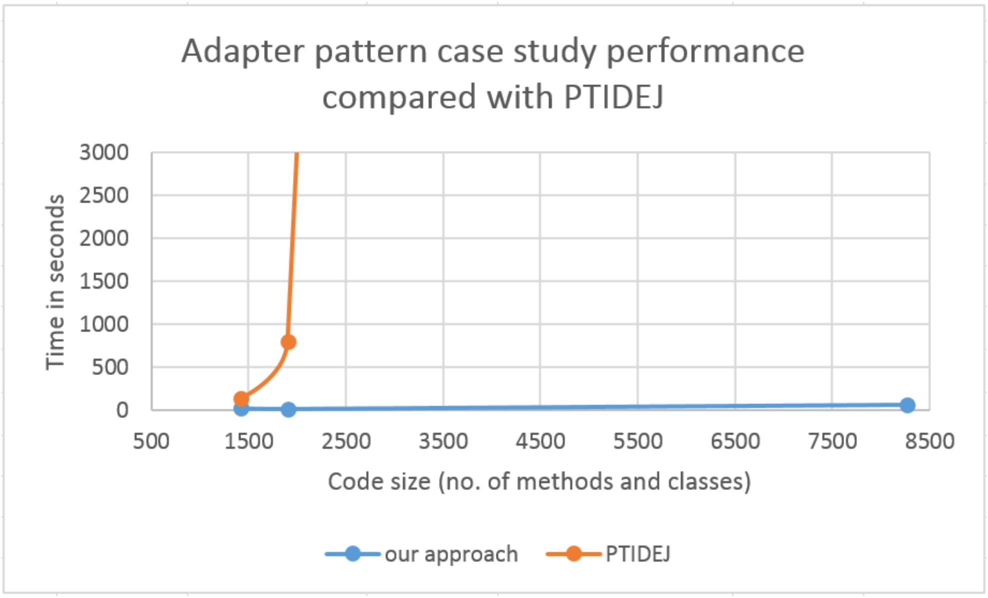


Figure 83: Adapter pattern case study performance compared with PTIDEJ

From Figure 83, it can be concluded that our approach significantly outperforms the Ptidej system. This is due to the fact that our approach is not considering the source codes components; it rather considers the relations among those components and only the relations which are relevant to the formulas representing the pattern structure. The source codes of the case studies are regarded as large-scale source codes. Therefore, it can be concluded that our approach is suitable to be used to detect patterns in the large-scale source codes.

Another performance comparison was made between our pattern detection approach and the DP-Miner pattern detection system. Figure 84 illustrates the performance of our pattern detection approach to detect three design patterns, namely Adapter, Composite, and Strategy in Java.AWT source code (708 components), with comparison to DP-Miner.

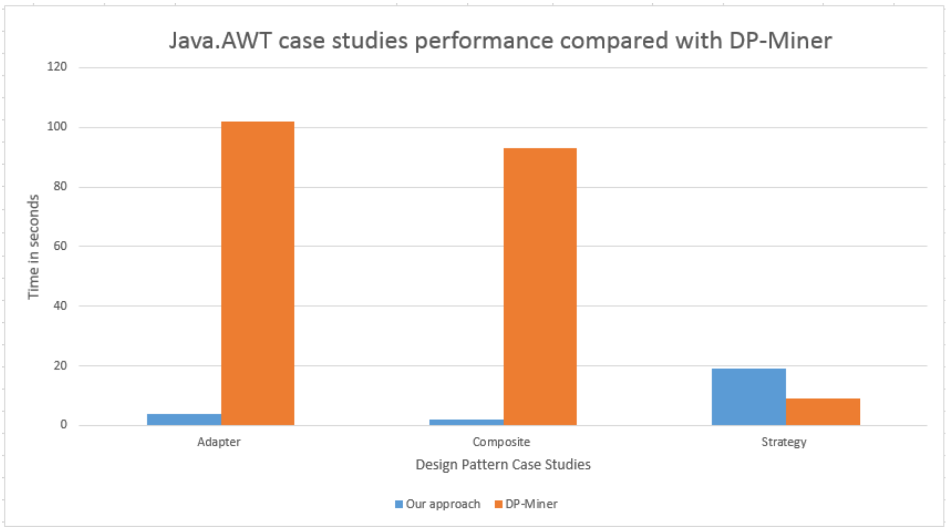


Figure 84: Java.AWT case study performance compared with DP-Miner

It can be seen from Figure 84 that our pattern detection approach considerably outperforms the DP-Miner in the performance of the detection process of Adapter and Composite patterns in Java.AWT. Although DP-Miner seems to perform better in the detection process of the Strategy pattern, the difference is not significant in time.

9.4 Summary

In conclusion, this chapter has described the evaluation of our pattern detection approach in comparison with other state-of-the-art pattern detection systems. First, our approach was evaluated using features analysis methodology. Second, our approach has been evaluated using the results of our approach case studies with comparison to other pattern detection systems case studies results by comparing the number of false positive instances. It can be concluded that our approach has been developed with considerations of the desirable features in pattern detection systems and it is competitive with the state-of-the-art pattern detection systems.

10 Conclusion

In conclusion, in this thesis, a new way of modelling security patterns was investigated and employed. Codecharts was used to formally and visually model security patterns. The investigation shows a number of benefits of using Codecharts. These benefits can be summarised in the following three points: 1) modelling secure relations among components; 2) modelling abstraction in patterns' descriptions; and 3) formality.

Additionally, using Codecharts for modelling security patterns highlights patterns variations, which come from the inconsistency of the descriptions of the same patterns in different security patterns catalogues. Using Codecharts allows for recognising the inconsistency and reasoning out the variations of the patterns. This would in turn enable the assurance of the same understanding among the people who experience security patterns in their work.

In addition, using Codecharts allows for further studies of patterns. One of these is the studying and reasoning of the relations among security patterns and with other well-known design patterns. Validating these relations helps in the overall understanding of the patterns as well as in the correct implementation of these patterns.

Despite the fact that security patterns formalisation has not been considered sufficiently by researchers, many researchers have introduced their approaches for design patterns formalisation. Based on the fact that security patterns are design patterns but for specific domains, these approaches have been compared and contrasted in terms of the formalisation direction adopted in each piece of research. The first drawback of these approaches is that they involve the transformation of UML diagrams to formal language. This introduces some types of information loss depending on the chosen formal language; whereas using Codecharts to model the security patterns from the patterns' description does not involve any transformation. Moreover, using Codecharts offers formal and visual outputs, unlike other approaches outputs which are often textual.

Furthermore, this thesis demonstrates a new pattern detection approach which aims to find, from the source code, instances of patterns that are represented as Codecharts. The approach, unlike other pattern detection systems, is developed for general patterns detection, which means that it can detect security, design, and/or user defined patterns.

The pattern detection approach proposed in this thesis was evaluated against the known features which have been outlined for pattern detection systems. In addition, the approach has been compared with a number of well-known pattern detection systems. Moreover, it has been implemented and integrated into the TTP Toolkit.

The approach was concluded to be competitive with peers' pattern detection systems in many features. Most importantly, the approach could compete in the feature of precision of the results. In addition, performance efficiency is a current problem that many pattern detection systems, which aim to find exact instances, suffer from. However, from the case studies conducted using the approach to detect exact instances of patterns in large scale source codes, the approach appears to be efficient in processing time in comparison with the existing most efficient pattern detection systems.

However, some disadvantages were experienced. First, Codecharts were able to model only decidable statements of the patterns descriptions. This means that they could not model some of the behavioural statements shown in patterns descriptions. Second, limited support and recognition of Codecharts stood as an obstacle for the use of Codecharts. Finally, the TTP Toolkit (the Codecharts tool support) had a lack of maintenance and a number of code bugs were not fixed.

This rest of this chapter shows a number of limitations which are worth mentioning here. The limitations focus on highlighting some obstacles and disadvantages of using Codecharts in modelling and detecting security patterns. In addition, the chapter suggests some directions for future work in order to take this research further and overcome the faced limitations.

10.1 LIMITATION

This section highlights a number of limitations which have been recognised along with the investigation in this research. These limitations might have attributed to the chosen way of modelling security patterns. They might also suggest ideas to take this research further in the future. However, these limitations have not been resolved in this research.

One limitation is in the used modelling notations which are Codecharts notations. Although Codecharts offer a competitive way for visually and formally modelling security patterns, it is limited only to modelling structural patterns. This is due to the LePUS3, which is the language of Codecharts. LePUS3 can only support specifying fully decidable statements. This limitation affects the modelling of some behavioural statements which exist in architectural security patterns. In addition, modelling behavioural security patterns was not possible.

The pattern detection approach demonstrated in this research was limited to static check of the detected instances of the looked-for patterns. Dynamic check was not possible in order to complete the checking of the instances. This limitation is due to the aforementioned limitation as Codecharts does not offer the modelling of behavioural statements of patterns. Furthermore, the TTP Toolkit verifier, which has been used to check the detected instances, is limited to static check of the source codes against the instances.

In addition to these limitations, although the TTP Toolkit, which is the tool support for Codecharts, is developed to deal with any given object-oriented source code, the latest version of it is limited to Java-based source codes. This fact shows the drop of support of the Codecharts and the TTP Toolkit. In addition, this limitation restricted the empirical case studies to Java-based source codes.

As this research uses the TTP Toolkit in the detection to model the patterns and to verify their instances, a number of code bugs in the TTP Toolkit were found during the use of it. The code bugs are mainly related to *Exclusive Operators (exclamation mark)*, in particular the *Right Exclusive Operator*. The TTP Toolkit verifier misunderstands it and considers it as the *Left Exclusive Operator.* So, it incorrectly verifies a modelling that has such use of it. In addition, the TTP Toolkit cannot model a Codecharts that has a 1-dim notation which has a relation from the notation or to the notation with any *Exclusive Operators (exclamation mark)* on the relation.

Finally, UML is widely used as a standard for any design modelling. Many researchers have introduced and built their work on patterns using UML. This has made its notations easy to be recognised and understood. However, although Codecharts offers some features which UML cannot, such as formality, without a need for transformation, Codecharts is not widely recognised. This might be due to the fact that some people consider it as UML-like diagrams and others consider it as a new language, although it has been more than 10 years since the introduction of its early versions. Another reason for Codecharts not being widely recognised is the lack of research on it. This includes extensions of it in order to model behavioural aspects. In addition, it includes the development and/or maintenance of tool supports.

10.2 FUTURE WORK

This research has focused on examining the use of Codecharts to formally model security patterns. It has also proposed a new novel approach for pattern detection using Codecharts and the TTP Toolkit. However, a number of research directions can be considered in order to improve the research results as well as to take this research further.

First, modelling the behavioural statements in the security patterns descriptions using Codecharts is the next demanding research direction. This direction of research needs to focus of extending LePUS3, the language of Codecharts, in order to enable it to accommodate modelling behavioural statements. However, the extension of LePUS3 might involve combining it with new behaviour notations from other modelling languages. It is important to be aware of the properties which LePUS3 has been built with.

In this research, the pattern detection approach employs the TTP Toolkit verifier to check and reason out the design conformance of the detected instances of a given pattern. However, the TTP Toolkit verifier only conducts a static checking of the given instances in the source code. Although static checking of the instances of patterns is the corner stone for any pattern detection system, dynamic checking is essential to determine whether the instances of the given pattern conform to the remaining behavioural properties of the pattern.

Dynamic checking needs to follow and rely on the static check of the instances of the given pattern. In addition, dynamic checking involves executing the source code and extracting the trace of method calls and objects creations in order to check the behaviours of the components which correspond to the given pattern's participants. The online and offline checking of the extracted trace should be available depending on the behavioural properties which have been modelled as the patterns behaviours.

To allow dynamic check, first, the direction of the extension LePUS3, the language of Codecharts, has to be carried out. Second, the TTP Toolkit verifier has to integrate new source code execution and trace extraction mechanism or probably import one of the existing ones. Finally, the pattern detection approach of this research has to be able to search through extracted execution trace of the source code for instances of the given pattern and link them to the detected instances after the static checking. This allows having two separate searching mechanisms, one for architectural and structural properties and another for behavioural properties, where the former relies on static checking and the latter relies on dynamic checking.

Future work may consider improving the presentation of the results of the pattern detection approach proposed in this research. One of these is introducing the ability to prepare a full printable report of the results of the pattern detection. Currently, the results of the pattern detection are shown in the left-hand side bar of the GUI of the TTP Toolkit. Although the results have been presented following the current categorisation of the TTP Toolkit GUI, a full printable report would enhance the readability of the results.

The pattern detection approach provides three types of outputs, which are a summary of the detection results (number of instances and time spent for the detection process), visualisations of instances as Codecharts, and assignments from the pattern's participants to the source code components. The visualisations of instances refer to the concrete entities in the source code. However, the relations among these entities are not referred to in the source code. Therefore, in the future, referring to the places of the relations in the source code by the line numbers would help with tracing the instances in the source code.

Finally, the existing pattern detection systems, such as DP-Miner, employ the visualisation of the instances of the looked-for pattern in order to ease the understanding of the detection results. Often, UML diagrams are applied for the results visualisations. In the pattern detection approach shown in this thesis, visualisations of the results of the detection have been considered and implemented. Codecharts have been used for the visualisation of the results. However, as has been discussed earlier, Codecharts notations are not as popular as UML diagrams. Therefore, in the future, beside the existing current feature of visualizing the outputs of the detection process, as in Codecharts, adding a feature of converting the Codecharts to UML diagrams would improve the readability of the outputs and enable the user to compare and understand the Codecharts.

References

[1] J. Yoder and J. Barcalow, ‘Architectural patterns for enabling application security’, *Urbana*, vol. 51, p. 61801, 1998.

[2] N. Yoshioka, H. Washizaki, and K. Maruyama, ‘A survey on security patterns’, *Prog. Inform.*, vol. 5, no. 5, pp. 35–47, 2008.

[3] Y.-G. Guéhéneuc, J.-Y. Guyomarc’h, and H. Sahraoui, ‘Improving design-pattern identification: a new approach and an exploratory study’, *Softw. Qual. J.*, vol. 18, no. 1, pp. 145–174, 2010.

[4] Y.-G. Guéhéneuc and G. Antoniol, ‘DeMIMA: A Multilayered Approach for Design Pattern Identification’, *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 667 –684, 2008.

[5] J. Dong, D. S. Lad, and Y. Zhao, ‘DP-Miner: Design Pattern Discovery Using Matrix’, in *Engineering of Computer-Based Systems, 2007. ECBS ’07. 14th Annual IEEE International Conference and Workshops on the*, 2007, pp. 371–380.

[6] F. A. Fontana, A. Caracciolo, and M. Zanoni, ‘DPB: A Benchmark for Design Pattern Detection Tools’, in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 235–244.

[7] Y.-G. Guéhéneuc, J.-Y. Guyomarc’h, D.-L. Huynh, O. Kaczor, N. Moha, S. Rached, and P. Team, ‘Ptidej: A Tool Suite’, 2005.

[8] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, ‘Design Pattern Detection Using Similarity Scoring’, *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896 –909, Nov. 2006.

[9] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security patterns. Integrating security and systems engineering.* 2006.

[10] J. Dong, T. Peng, and Y. Zhao, ‘Model Checking Security Pattern Compositions’, in *Seventh International Conference on Quality Software, 2007. QSIC ’07*, 2007, pp. 80 –89.

[11] J. K. Y. Ng, Y. G. Gu’eh’eneuc, and G. Antoniol, ‘Identification of behavioural and creational design motifs through dynamic analysis’, *J. Softw. Maint. Evol. Res. Pract.*, vol. 22, no. 8, pp. 597–627, 2010.

[12] D. Serrano, A. Maa, and A.-D. Sotirious, ‘Towards Precise Security Patterns’, in *19th International Workshop on Database and Expert Systems Application, 2008. DEXA ’08*, 2008, pp. 287 –291.

[13] J. Nicholson and A. Eden, ‘TTP Toolkit - Home - Object-oriented design, visual modelling, formal specification, automated verification, reverse engineering, design mining, traceability, scalability’, May-2009. [Online]. Available: http://ttp.essex.ac.uk/. [Accessed: 30-Dec-2013].

[14] M. Vujoševic-Janicic and D. Tošic, ‘The role of programming paradigms in the first programming courses’, *Teach. Math. XI*, vol. 2, pp. 63–83, 2008.

[15] G. Booch, ‘Object-oriented development’, *Softw. Eng. IEEE Trans. On*, no. 2, pp. 211–221, 1986.

[16] P. Wegner, ‘Concepts and paradigms of object-oriented programming’, *ACM SIGPLAN OOPS Messenger*, vol. 1, no. 1, pp. 7–87, 1990.

[17] J. Martin and J. J. Odell, *Object-oriented methods*. Prentice hall PTR, 1994.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1995.

[19] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java*. Prentice Hall, 2004.

[20] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, ‘Software complexity and maintenance costs’, *Commun. ACM*, vol. 36, no. 11, pp. 81–94, 1993.

[21] W. Meng, J. Rilling, Y. Zhang, R. Witte, and P. Charland, ‘An ontological software comprehension process model’, in *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM 2006). October 1st, Genoa, Italy*, 2006.

[22] M. Berón, P. Henriques, M. J. Pereira, and R. Uzal, ‘Program inspection to interconnect behavioral and operational view for program comprehension’, 2007.

[23] M.-A. Storey, ‘Theories, methods and tools in program comprehension: Past, present and future’, in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, 2005, pp. 181–191.

[24] R. Quirk and D. Summers, *Longman dictionary of contemporary English*. Longman, 1987.

[25] B. Blakeley, *Introduction to Security Design Patterns*. Open Group, 2004.

[26] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*, 2nd ed. Prentice Hall Professional, 2003.

[27] R. Wassermann and B. H. C. Cheng, ‘Security patterns’, in *Michigan State University, PLoP Conf*, 2003.

[28] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt, ‘Security patterns repository version 1.0’, *Def. Adv. Res. Proj. Agency DARPA Contract F30602-01-C-0164 Wash. DC*, 2002.

[29] S. Konrad, B. H. C. Cheng, L. A. Campbell, and R. Wassermann, ‘Using security patterns to model and analyze security requirements’, *Requir. Eng. High Assur. Syst. RHAS03*, p. 11, 2003.

[30] M. Bunke and K. Sohr, ‘An Architecture-Centric Approach to Detecting Security Patterns in Software’, in *Engineering Secure Software and Systems*, vol. 6542, Ú. Erlingsson, R. Wieringa, and N. Zannone, Eds. Springer Berlin / Heidelberg, 2011, pp. 156–166.

[31] A. K. Alvi and M. Zulkernine, ‘A Natural Classification Scheme for Software Security Patterns’, in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, 2011, pp. 113 –120.

[32] H. Washizaki, E. B. Fernandez, K. Maruyama, A. Kubo, and N. Yoshioka, ‘Improving the Classification of Security Patterns’, in *Database and Expert Systems Application, 2009. DEXA ’09. 20th International Workshop on*, 2009, pp. 165 –170.

[33] M. Hafiz, P. Adamczyk, and R. E. Johnson, ‘Organizing Security Patterns’, *Softw. IEEE*, vol. 24, no. 4, pp. 52 –60, Aug. 2007.

[34] M. Weiss and H. Mouratidis, ‘Selecting Security Patterns that Fulfill Security Requirements’, in *International Requirements Engineering, 2008. RE ’08. 16th IEEE*, 2008, pp. 169 –172.

[35] G. McGraw, ‘Software security’, *IEEE Secur. Priv.*, vol. 2, no. 2, pp. 80–83, 2004.

[36] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*, 2nd ed. Cambridge University Press, 2004.

[37] J. Barwise and J. Etchemendy, *The Language of First-Order Logic, Including the Macintosh Program Tarski’s World 4.0: Including the Macintosh Programme, Tarski’s World 4.0*, 3 edition. Stanford, Calif: The Center for the Study of Language and Information Publications, 1993.

[38] A. H. Eden and J. Nicholson, *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs*. John Wiley & Sons, 2011.

[39] E. Gasparis, J. Nicholson, and A. Eden, ‘LePUS3: An Object-Oriented Design Description Language’, in *Diagrammatic Representation and Inference*, vol. 5223, G. Stapleton, J. Howse, and J. Lee, Eds. Springer Berlin / Heidelberg, 2008, pp. 364–367.

[40] A. Eden, ‘A visual formalism for object-oriented architecture’, *Proc. Integr. Des. Process Technol. IDPT-2002*, 2002.

[41] B. Potter, D. Till, and J. Sinclair, *An introduction to formal specification and Z*. Prentice Hall PTR, 1996.

[42] P. Freeman, *Software systems principles: a survey*. Science Research Associates, 1975.

[43] V. D’silva, D. Kroening, and G. Weissenbacher, ‘A survey of automated techniques for formal software verification’, *Comput.-Aided Des. Integr. Circuits Syst. IEEE Trans. On*, vol. 27, no. 7, pp. 1165–1178, 2008.

[44] G. J. Holzmann, ‘The model checker SPIN’, *Softw. Eng. IEEE Trans. On*, vol. 23, no. 5, pp. 279–295, 1997.

[45] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, ‘KRATOS– A Software Model Checker for SystemC’, in *Computer Aided Verification*, 2011, pp. 310–316.

[46] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser, ‘Tool-supported program abstraction for finite-state verification’, in *Proceedings of the 23rd international conference on software engineering*, 2001, pp. 177–187.

[47] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng, ‘Bandera: Extracting finite-state models from Java source code’, in *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, 2000, pp. 439–448.

[48] Ptidej Team, ‘Research — Ptidej Team’, 2013. [Online]. Available: http://www.ptidej.net/research/. [Accessed: 28-Feb-2014].

[49] S. Malakuti, C. Bockisch, and M. Aksit, ‘Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software’, in *20th International Symposium on Software Reliability Engineering, 2009. ISSRE ’09*, 2009, pp. 31 –40.

[50] T.-D. Cao, T.-T. Phan-Quang, P. Felix, and R. Castanet, ‘Automated Runtime Verification for Web Services’, in *2010 IEEE International Conference on Web Services (ICWS)*, 2010, pp. 76 –82.

[51] J. Shao, F. Deng, H. Liu, Q. Wang, and H. Mei, ‘Lazy Runtime Verification for Constraints on Interacting Objects’, in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010, pp. 242 –251.

[52] J. B. Wordsworth, *Software development with Z: a practical approach to formal methods in software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[53] W. E. McUmber and B. H. Cheng, ‘A general framework for formalizing UML with formal languages’, in *Proceedings of the 23rd international conference on Software engineering*, 2001, pp. 433–442.

[54] B. Selic, ‘A systematic approach to domain-specific language design using UML’, in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC’07. 10th IEEE International Symposium on*, 2007, pp. 2–9.

[55] W. Kadir, W. M. Nasir, and R. Mohamad, ‘Formalization of UML class diagram using description logics’, 2010.

[56] W. Yan and Y. Du, ‘Research on reverse engineering from formal models to UML models’, in *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, 2010, pp. 406–411.

[57] H. M. Chavez and W. Shen, 'Formalization of UML Composition in OCL', in *2012 IEEE/ACIS 11th International Conference on Computer and Information Science (ICIS)*, 2012, pp. 675–680.

[58] B. Zhou, J. Lu, Z. Wang, Y. Zhang, and Z. Miao, 'Formalizing fuzzy UML class diagrams with fuzzy description logics', in *Intelligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, 2009, vol. 1, pp. 171–174.

[59] S. Sengupta and S. Bhattacharya, 'Formalization of UML diagrams and their consistency verification: A Z notation based approach', in *Proceedings of the 1st India software engineering conference*, New York, NY, USA, 2008, pp. 151–152.

[60] A. M. Mostafa, M. A. Ismail, H. El-Bolok, and E. M. Saad, 'Toward a Formalization of UML2.0 Metamodel using Z Specifications', in *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007*, 2007, vol. 1, pp. 694–701.

[61] Z. Zhihong and Z. Mingtian, 'Some considerations in formalizing UML class diagrams with description logics', in *Robotics, Intelligent Systems and Signal Processing, 2003. Proceedings. 2003 IEEE International Conference on*, 2003, vol. 1, pp. 111–115.

[62] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, 'A formal framework for reasoning on UML class diagrams', in *Foundations of Intelligent Systems*, Springer, 2002, pp. 503–513.

[63] M. Gogolla and M. Richters, 'Expressing UML Class Diagrams Properties with OCL', in *Object Modeling with the OCL*, T. Clark and J. Warmer, Eds. Springer Berlin Heidelberg, 2002, pp. 85–114.

[64] D. Berardi, D. Calvanese, and G. De Giacomo, 'Reasoning on UML class diagrams using description logic based systems', in *Proc. of the KI'2001 Workshop on Applications of Description Logics*, 2001, vol. 44.

[65] Y. Ledru, 'Using Jaza to Animate RoZ Specifications of UML Class Diagrams', in *Software Engineering Workshop, 2006. SEW '06. 30th Annual IEEE/NASA*, 2006, pp. 253–262.

[66] M. Bittner and F. Kammuller, 'Translating Fusion/UML to Object-Z', in *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings*, 2003, pp. 49–50.

[67] E. Sekerinski and R. Zurob, 'Translating statecharts to B', in *Integrated Formal Methods*, 2002, pp. 128–144.

[68] X. Li, Z. Liu, and H. Jifeng, 'A formal semantics of UML sequence diagram', in *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, 2004, pp. 168–177.

[69] M. Y. Ng and M. Butler, 'Towards formalizing UML state diagrams in CSP', in *First International Conference on Software Engineering and Formal Methods, 2003.Proceedings*, 2003, pp. 138–147.

[70] S.-K. Kim and C. David, 'Formalizing the UML class diagram using Object-Z', in *«UML»'99—The Unified Modeling Language*, Springer, 1999, pp. 83–98.

[71] OMG, 'OMG Object Constraint Language (OCL)', Jan. 2012.

[72] I. S. Bajwa, B. Bordbar, and M. G. Lee, 'OCL Constraints Generation from Natural Language Specification', in *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*, 2010, pp. 204–213.

[73] A. Shaikh, U. K. Wiil, and N. Memon, 'Evaluation of tools and slicing techniques for efficient verification of UML/OCL class diagrams', *Adv. Softw. Eng.*, vol. 2011, p. 5, 2011.

[74] F. Hilken, P. Niemann, R. Wille, and M. Gogolla, 'Towards a Base Model for UML and OCL Verification⋆', in *11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014*, 2014, p. 59.

[75] J. Cabot, R. Clarisó, and D. Riera, 'On the verification of UML/OCL class diagrams using constraint programming', *J. Syst. Softw.*, vol. 93, pp. 1–23, 2014.

[76] G. Rull, C. Farré, A. Queralt, E. Teniente, and T. Urpí, 'AuRUS: explaining the validation of UML/OCL conceptual schemas', *Softw. Syst. Model.*, pp. 1–28, 2013.

[77] M. Balaban and A. Maraee, 'Finite satisfiability of UML class diagrams with constrained class hierarchy', *ACM Trans. Softw. Eng. Methodol. TOSEM*, vol. 22, no. 3, p. 24, 2013.

[78] R. Wille, M. Soeken, and R. Drechsler, 'Debugging of inconsistent UML/OCL models', in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 1078–1083.

[79] A. Queralt and E. Teniente, 'Verification and validation of UML conceptual schemas with OCL constraints', *ACM Trans. Softw. Eng. Methodol. TOSEM*, vol. 21, no. 2, p. 13, 2012.

[80] S. Flake and W. Mueller, 'An OCL extension for real-time constraints', in *Object Modeling with the OCL*, Springer, 2002, pp. 150–171.

[81] A. Evans, R. France, K. Lano, and B. Rumpe, 'Developing the UML as a formal modelling notation', in *Proc. UML'98, LNCS*, 1998, vol. 1618.

[82] F. Baader, *The description logic handbook: theory, implementation, and applications*. Cambridge: Cambridge University Press, 2003.

[83] I. Horrocks, U. Sattler, and S. Tobies, 'Practical reasoning for expressive description logics', in *Logic for Programming and Automated Reasoning*, 1999, pp. 161–180.

[84] V. Haarslev and R. Müller, 'RACER system description', in *Automated Reasoning*, Springer, 2001, pp. 701–705.

[85] B. N. Grosof, I. Horrocks, R. Volz, and S. Decker, 'Description logic programs: Combining logic programs with description logic', in *Proceedings of the 12th international conference on World Wide Web*, 2003, pp. 48–57.

[86] G. Smith, *The Object-Z specification language*, vol. 101. Citeseer, 2000.

[87] F. A. Fontana, M. Zanoni, and S. Maggioni, 'Using Design Pattern Clues to Improve the Precision of Design Pattern Detection Tools.', *J. Object Technol.*, vol. 10, no. 4, pp. 1–31, 2011.

[88] F. Arcelli Fontana and M. Zanoni, 'A tool for design pattern detection and software architecture reconstruction', *Inf. Sci.*, vol. 181, no. 7, pp. 1306–1324, Apr. 2011.

[89] Y.-G. Guéhéneuc, 'PMARt: Pattern-like micro architecture repository', *Proc. 1st Eur. Focus Group Pattern Repos.*, 2007.

[90] F. Arcelli, S. Masiero, and C. Raibulet, 'Elemental Design Patterns Recognition In Java', in *13th IEEE International Workshop on Software Technology and Engineering Practice, 2005*, 2005, pp. 196–205.

[91] J. McC Smith and D. Stotts, 'Elemental Design Patterns: A formal semantics for composition of OO software architecture', in *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, 2002, pp. 183–190.

[92] Y.-G. Guéhéneuc and N. Jussien, 'Using explanations for design-patterns identification', in *IJCAI*, 2001, vol. 1, pp. 57–64.

[93] M. VanHilst and E. B. Fernandez, 'Reverse engineering to detect security patterns in code', in *Proc. of 1st International Workshop on Software Patterns and Quality. Information Processing Society of Japan (December 2007)*, 2007.

[94] S. Uchiyama, A. Kubo, H. Washizaki, and Y. Fukazawa, 'Detecting Design Patterns in Object-Oriented Program Source Code by using Metrics and Machine Learning Special Issue-Software Design Pattern', *J. Softw. Eng. Appl. Sci. Res. Publ.*, vol. 7, 2014.

[95] M. Gupta, A. Pande, and A. K. Tripathi, 'Design patterns detection using SOP expressions for graphs', *SIGSOFT Softw Eng Notes*, vol. 36, no. 1, pp. 1–5, Jan. 2011.

[96] G. Rasool and P. Mader, 'Flexible design pattern detection based on feature types', in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, Washington, DC, USA, 2011, pp. 243–252.

[97] N. Pettersson, W. Lowe, and J. Nivre, 'Evaluation of accuracy in design pattern occurrence detection', *Softw. Eng. IEEE Trans. On*, vol. 36, no. 4, pp. 575–590, 2010.

[98] J. M. Smith and D. Stotts, 'SPQR: flexible automated design pattern extraction from source code', in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, 2003, pp. 215–224.

[99] J. M. Smith, 'SPQR: formal foundations and practical support for the automated detection of design patterns from source code', Citeseer, 2005.

[100] N. Shi and R. A. Olsson, 'Reverse engineering of design patterns for high performance computing', in *Proceedings of the 2005 Workshop on Patterns in High Performance Computing*, 2005.

[101] P. Wegrzynowicz and K. Stencel, 'Relaxing queries to detect variants of design patterns', in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, 2013, pp. 1571–1578.

[102] Y.-G. Guéhéneuc, 'Ptidej: Promoting patterns with patterns', in *European Conference on Object Oriented Programming, workshop on Building a System with Patterns, Glasgow, Scotland*, 2005.

[103] V. R. Basili, 'Software modeling and measurement: the Goal/Question/Metric paradigm', 1992.

[104] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, 'A tactic-centric approach for automating traceability of quality concerns', in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 639–649.

[105] M. Mirakhorli and J. Cleland-Huang, 'Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance', in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 123–132.

[106] R. Wassermann and B. H. C. Cheng, 'Security Patterns', presented at the Pattern Languages of Programs—PLoP 2003, Robert Allerton Park, MI, 2003.

[107] F. L. Brown, J. DiVietri, G. D. de Villegas, and E. B. Fernandez, 'The authenticator pattern', *Proc. Pattern Lang. Programs PloP'99*, pp. 15–18, 1999.

[108] A. H. Eden, E. Gasparis, and J. Nicholson, 'LePUS3 and Class-Z Reference Manual', University of Essex, Tech. Rep, 2007.

[109] J. Yoder and J. Barcalow, 'Architectural Patterns for Enabling Application Security', in *Pattern languages of program design 4*, B. Foote, N. Harrison, and H. Rohnert, Eds. Addison-Wesley, 2000.

[110] I. Bayley and H. Zhu, 'Formal specification of the variants and behavioural features of design patterns', *J. Syst. Softw.*, vol. 83, no. 2, pp. 209–221, Feb. 2010.

[111] I. Bayley and H. Zhu, 'Formalising design patterns in predicate logic', in *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, 2007, pp. 25–36.

[112] A. Hachemi and M. Ahmed-Nacer, ‘Automatic Extraction of Relationships Among Software Patterns’, in *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering*, Springer, 2015, pp. 575–580.

[113] M. Hafiz, P. Adamczyk, and R. E. Johnson, ‘Towards an Organization of Security Patterns’, *Volume*, vol. 24, pp. 52–60, 2007.

[114] A. Kubo, H. Washizaki, and Y. Fukazawa, ‘Extracting relations among security patterns’, in *Submitted to 1st International Workshop on Software Patterns and Quality (SPAQu’07)*, 2007.

[115] I. Bayley and H. Zhu, ‘A formal language of pattern composition’, in *Proceedings of the 2nd international conference on pervasive patterns (PATTERNS 2010), XPS (Xpert Publishing Services), Lisbon, Portugal*, 2010, pp. 1–6.

[116] Oracle Java Technology, ‘JAAS Reference Guide’, 2011. [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html. [Accessed: 06-Jul-2012].

[117] Oracle Java Technology, ‘JAAS Authentication Tutorial’, 2011. [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html#Sample. [Accessed: 16-Jul-2012].

[118] Log4J, ‘Overview - Apache Log4j 2’, 2014. [Online]. Available: http://logging.apache.org/log4j/2.x/manual/index.html. [Accessed: 25-Sep-2014].

[119] C. Gulcu, ‘Short introduction to log4j’, 2002.

[120] Apache Struts, ‘The Apache Software Foundation’. [Online]. Available: http://www.apache.org/. [Accessed: 16-Mar-2015].

[121] C.-H. Chang, C.-W. Lu, W. C. Chu, N.-L. Hsueh, and C.-S. Koong, ‘A Case Study of Pattern-based Software Framework to Improve the Quality of Software Development’, in *Proceedings of the 2009 ACM Symposium on Applied Computing*, New York, NY, USA, 2009, pp. 443–447.

[122] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[123] D. S. Johnson, ‘A Brief History of NP-Completeness, 1954–2012’, *Doc. Math.*, pp. 359–376, 2012.

[124] http://jrefactory.sourceforge.net/, ‘JRefactory’, May-2004. [Online]. Available: http://jrefactory.sourceforge.net/. [Accessed: 25-Sep-2014].

[125] http://www.jhotdraw.org/, ‘JHotDraw Start Page’, *JHotDraw Start Page*, 11-Aug-2007. [Online]. Available: http://www.jhotdraw.org/. [Accessed: 25-Sep-2014].

[126] Wolfram Kaiser, ‘Become a programming Picasso with JHotDraw’, *JavaWorld*, 16-Feb-2001. [Online]. Available: http://www.javaworld.com/article/2074997/swing-gui-programming/become-a-programming-picasso-with-jhotdraw.html. [Accessed: 25-Sep-2014].

[127] R. Margea and C. Margea, ‘Open Source Approach to Project Management Tools’, *Inform. Econ.*, vol. 15, no. 1, pp. 196–206, 2011.

[128] http://www.ganttproject.biz/, ‘GanttProject: free desktop project management app’, Sep-2014. [Online]. Available: http://www.ganttproject.biz/. [Accessed: 25-Sep-2014].

[129] http://junit.org/, ‘JUnit - About’, 25-Sep-2014. [Online]. Available: http://junit.org/. [Accessed: 25-Sep-2014].

[130] Elliotte Rusty Harold, ‘An early look at JUnit 4’, 13-Sep-2005. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-junit4/. [Accessed: 25-Sep-2014].

[131] Chad McHenry and Robert Thomas, ‘JSettlers’, *SourceForge*, 15-Apr-2013. [Online]. Available: http://sourceforge.net/projects/jsettlers/. [Accessed: 12-Mar-2015].

[132] G. Rasool and D. Streitfdert, ‘A Survey on Design Pattern Recovery Techniques.’, *Int. J. Comput. Sci. Issues IJCSI*, vol. 8, no. 6, 2011.

[133] G. Rasool, P. Maeder, and I. Philippow, ‘Evaluation of design pattern recovery tools’, *Procedia Comput. Sci.*, vol. 3, pp. 813–819, 2011.

[134] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, ‘Design pattern recovery through visual language parsing and source code analysis’, *J. Syst. Softw.*, vol. 82, no. 7, pp. 1177–1193, Jul. 2009.

Appendices

APPENDIX A: More Security patterns in Codecharts

- Check Point Pattern

Pattern description catalogue:[106]

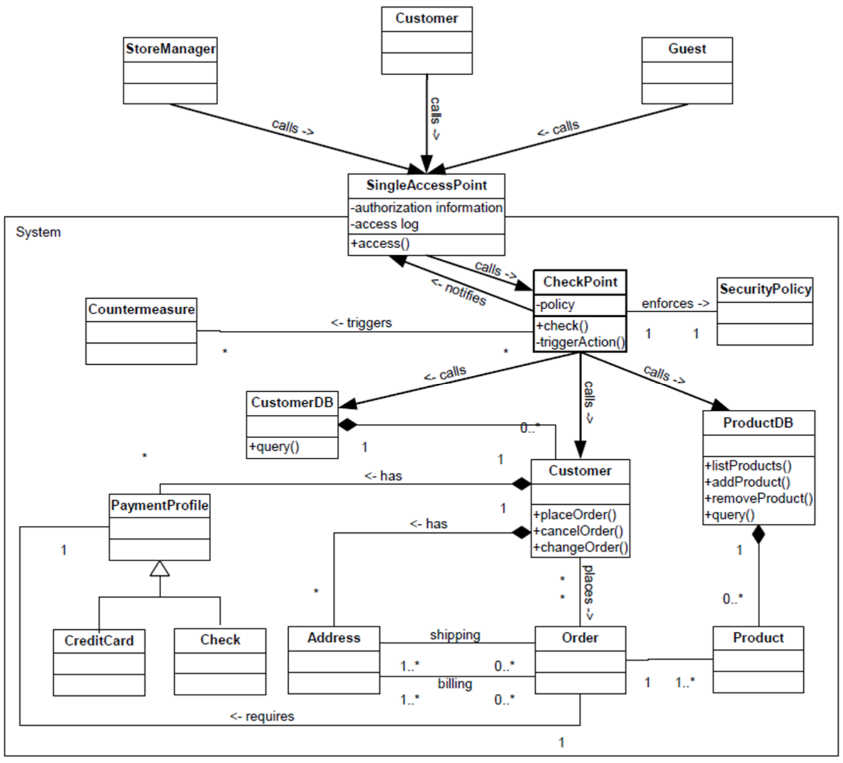


Figure 85: UML class diagram: Check Point Pattern [27]

- Secure Logger

Pattern description catalogue: [26]

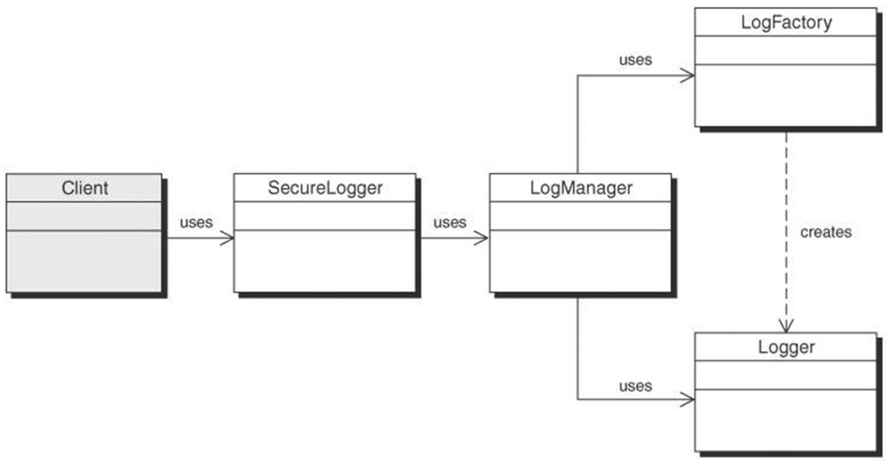


Figure 86: UML class diagram: Secure Logger Pattern [26]

SecureLoggerPattern

$Client, Handler, Logger, SecureLogger, logFactory : \mathbb{CLASS}$
$logManager : \mathbb{CLASS}$
$Request, createLogger, getHandler, getLogger, log : \mathbb{SIGNATURE}$

$Call(Request \otimes Client, log \otimes SecureLogger)$
$Call(createLogger \otimes logManager, getLogger \otimes logFactory)$
$Call(log \otimes SecureLogger, log \otimes logManager)$
$Create(getHandler \otimes logFactory, Handler)$
$Create(getLogger \otimes logFactory, Logger)$
$Forward(log \otimes logManager, log \otimes Logger)$
$Member(Client, SecureLogger)$
$Member(Logger, Handler)$
$Member(logFactory, Handler)$

Figure 87: LePUS3 and Class-Z schema of Secure Logger Pattern

- Role pattern

Pattern description catalogue: [106]

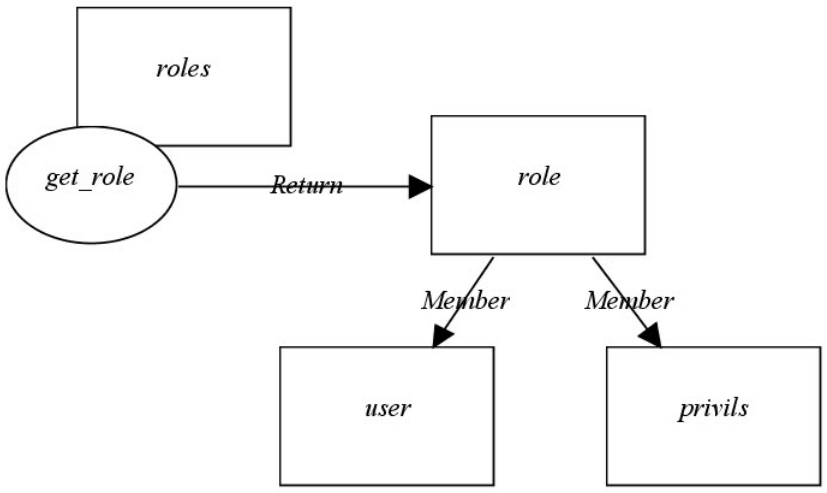


Figure 88: Codecharts: Role pattern

- Secure Architecture

This pattern combines Single Access Point, Check Point, and Role.

Pattern description catalogue: [106]

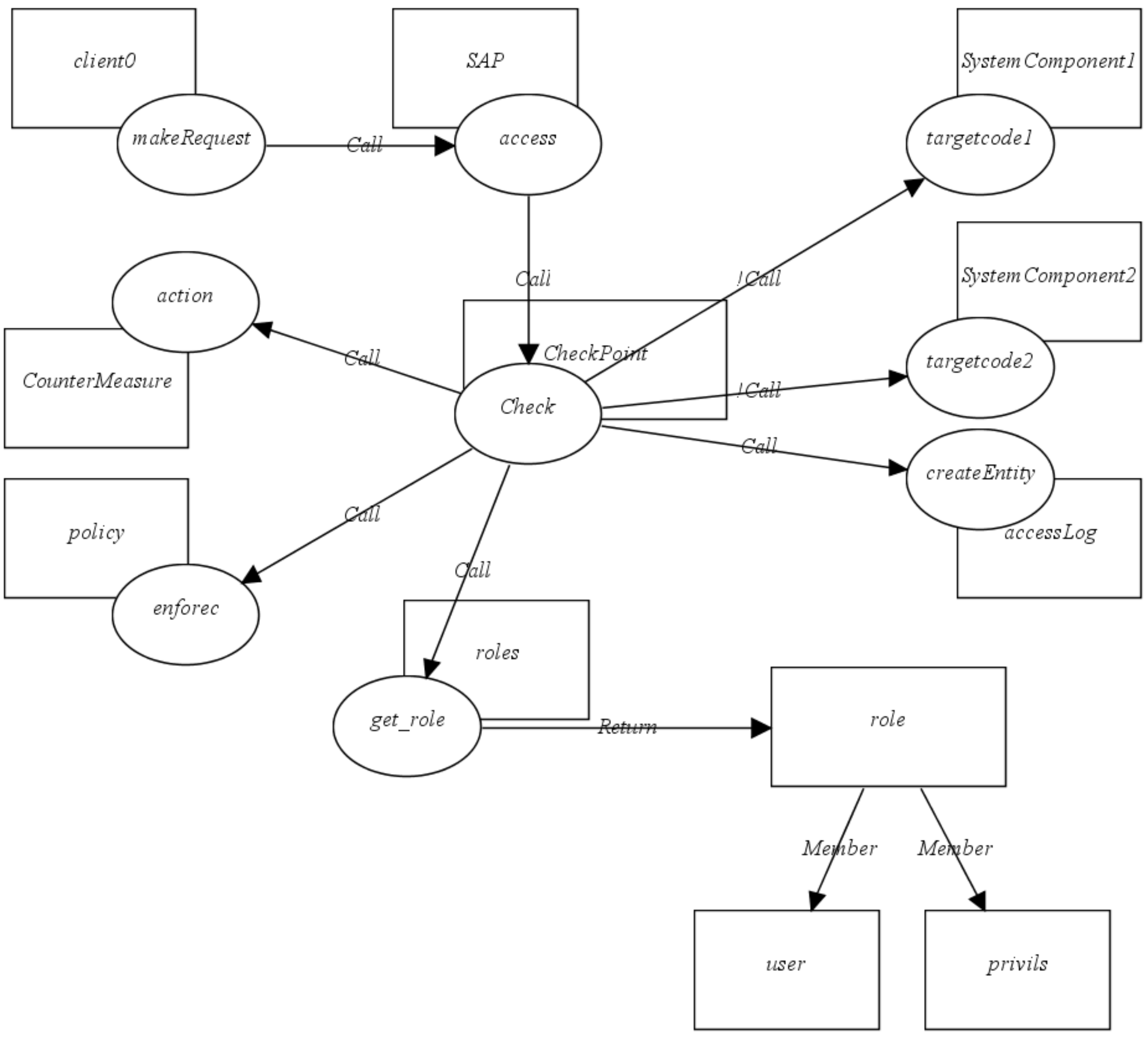


Figure 89: Codecharts: Secure Architecture

- Role and Session

Pattern description catalogue: [106]

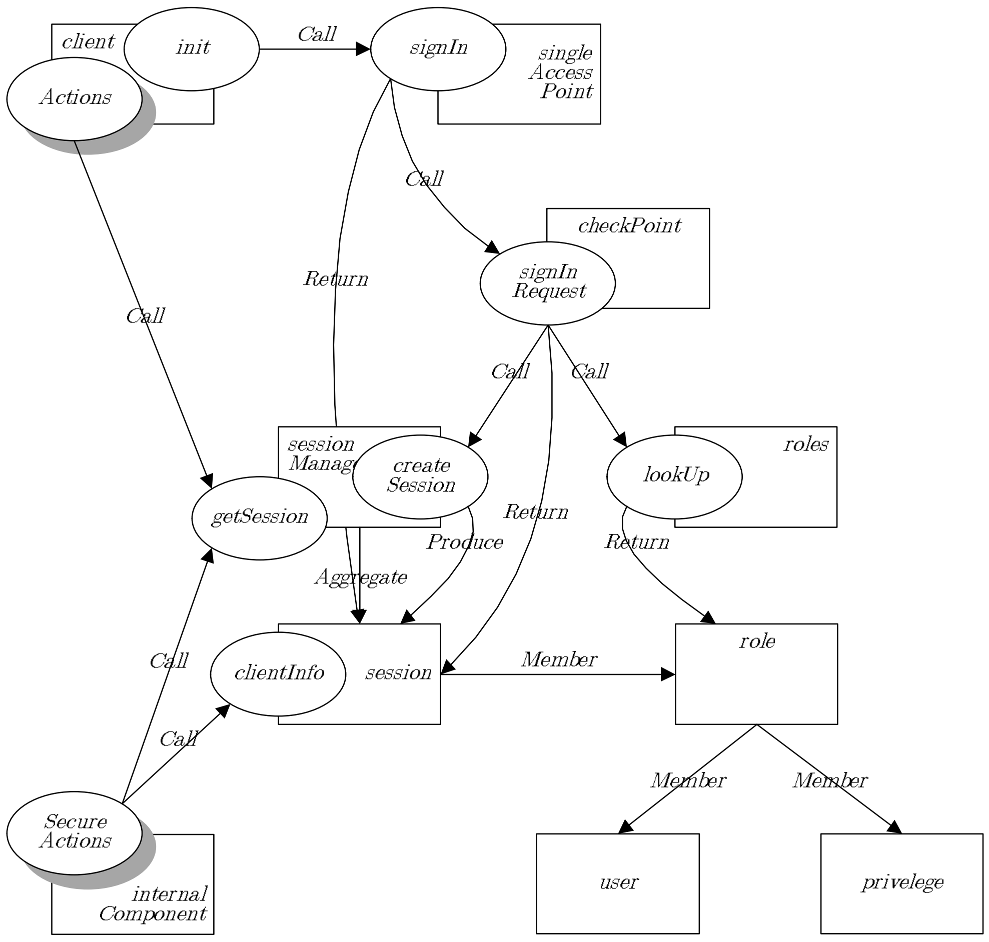


Figure 90: Codecharts: Role and Session

- Authentication Enforcer Pattern

Pattern description catalogue:[26]

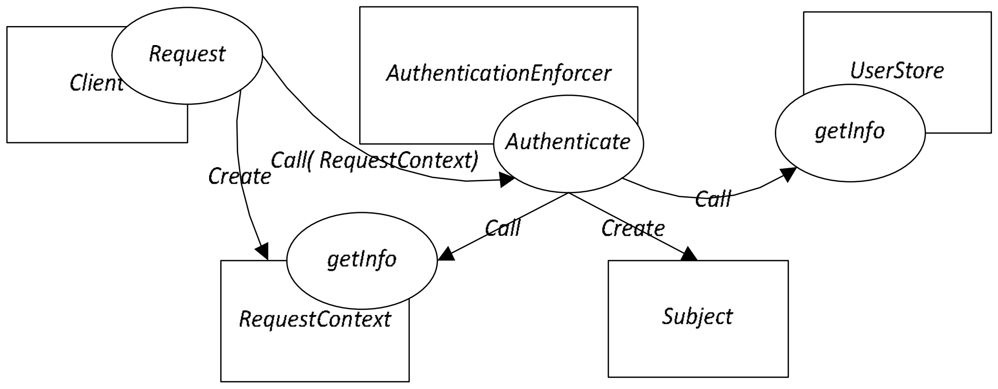


Figure 91: Codecharts: Authentication Enforcer Pattern

- Account Lockout Pattern

Pattern description catalogue: [28]

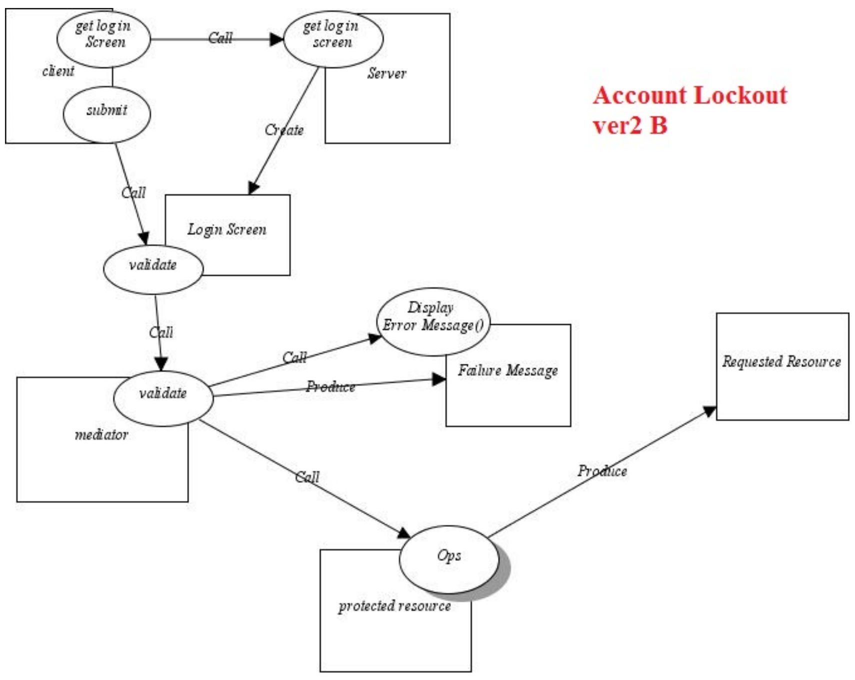


Figure 92: Codecharts: Account Lockout Pattern

- Authenticated Session Pattern

Pattern description catalogue: [28]

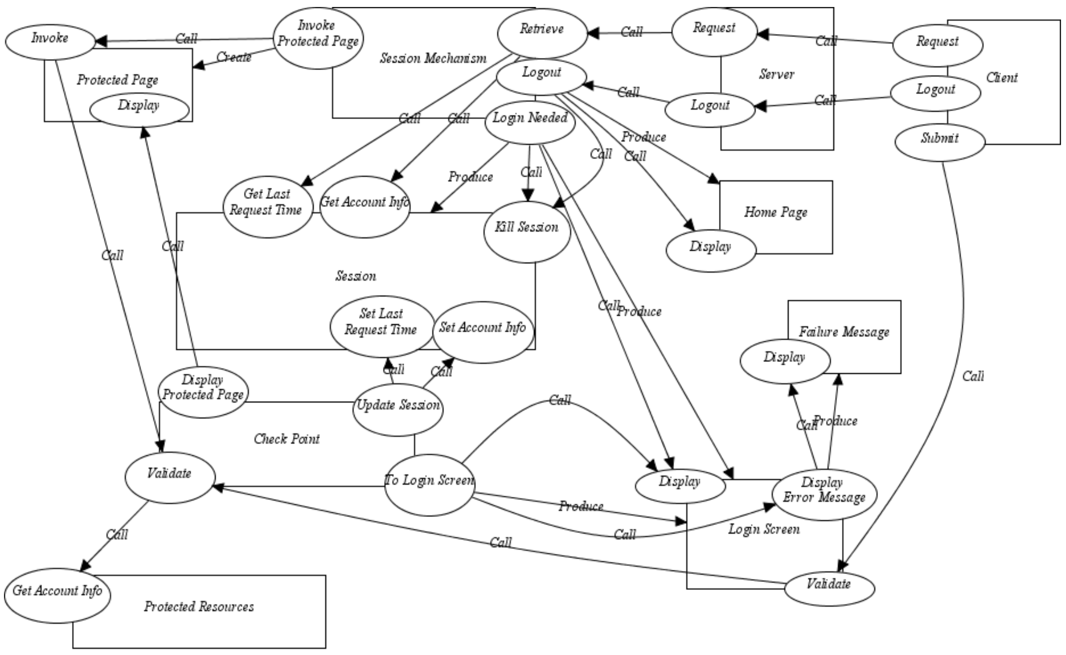


Figure 93: Codecharts: Authenticated Session Pattern

- Client Data Storage Pattern

Pattern description catalogue: [28]

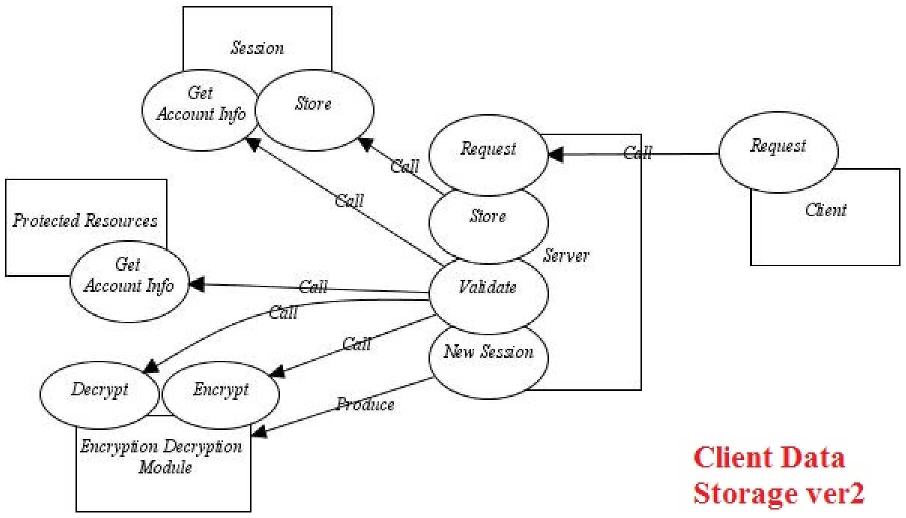


Figure 94: Codecharts: Client Data Storage pattern

- Network Address blacklist Pattern

Pattern description catalogue: [28]

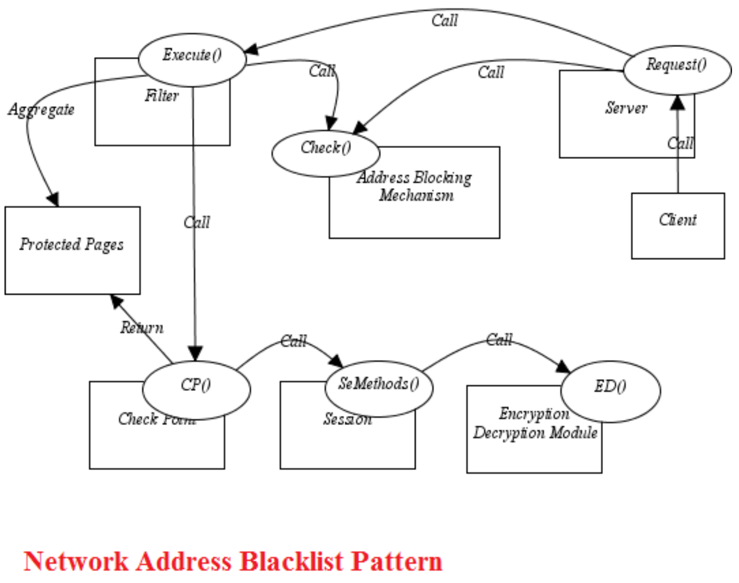


Figure 95: Codecharts: Network Address blacklist pattern

- Password Authentication Pattern

Pattern description catalogue: [28]

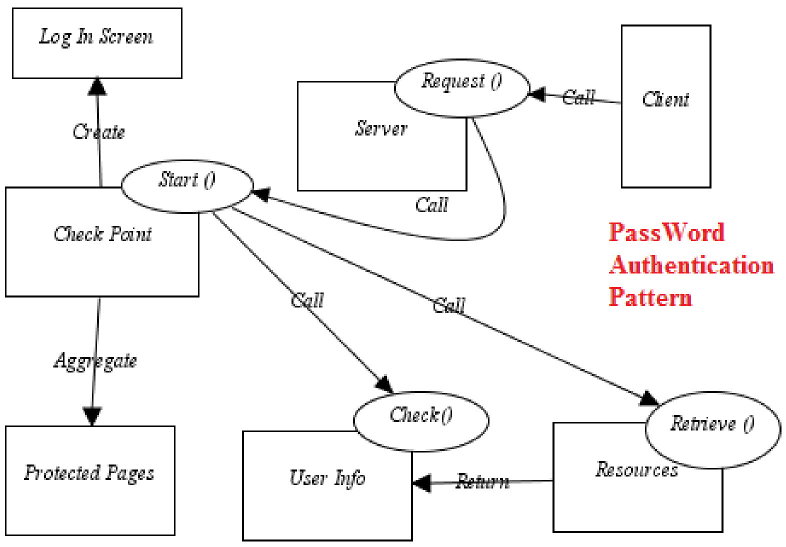


Figure 96: Codecharts: Password Authentication pattern

- Password Propagation Pattern

Pattern description catalogue: [28]

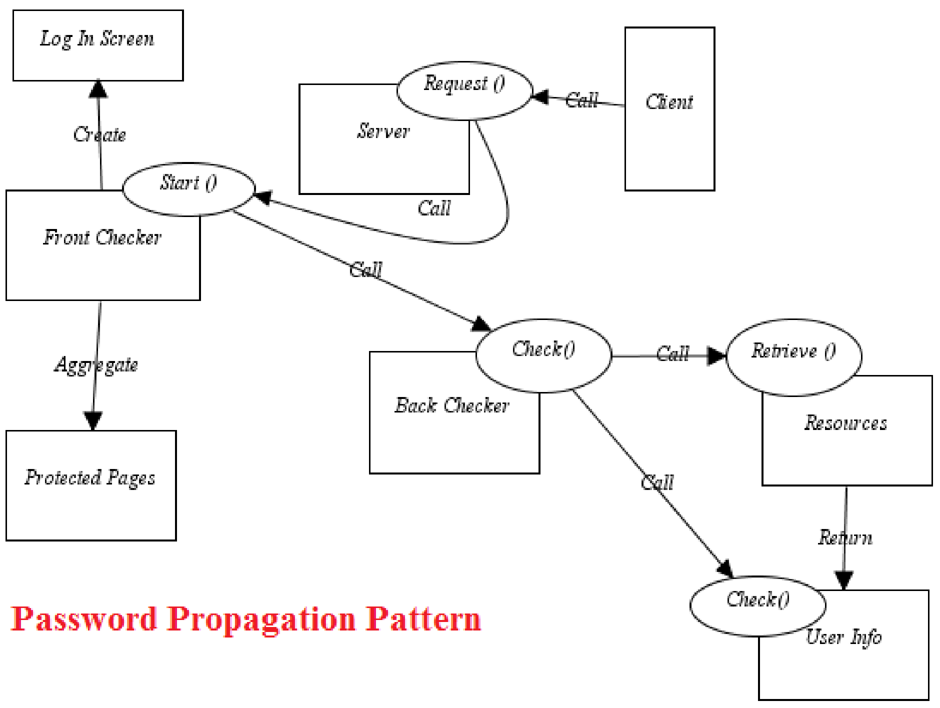


Figure 97: Codecharts: Password Propagation pattern

- Full View with Error pattern

Pattern description catalogue: [9]

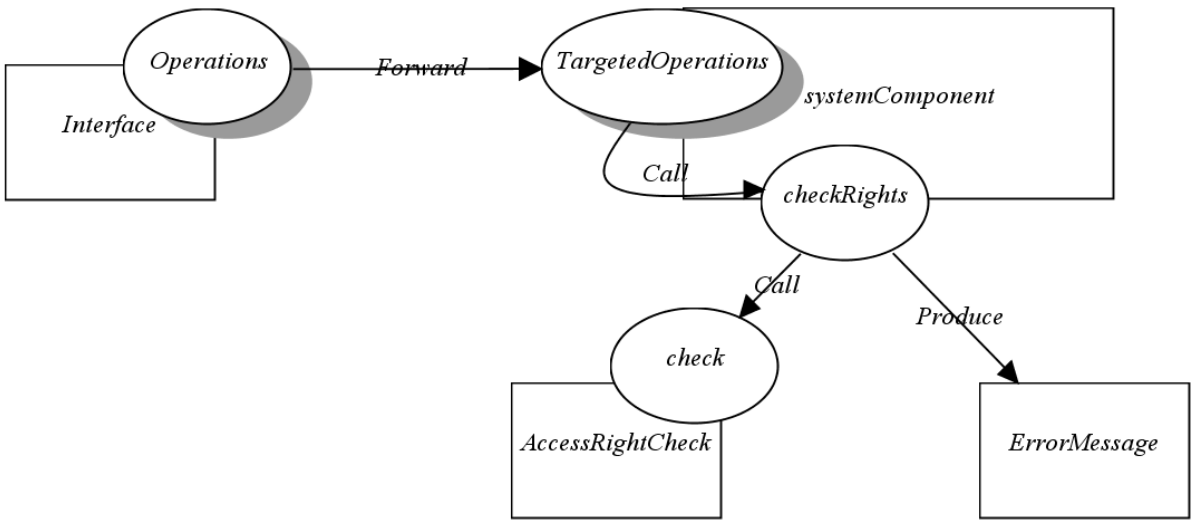


Figure 98: Codecharts: Full View with Error pattern

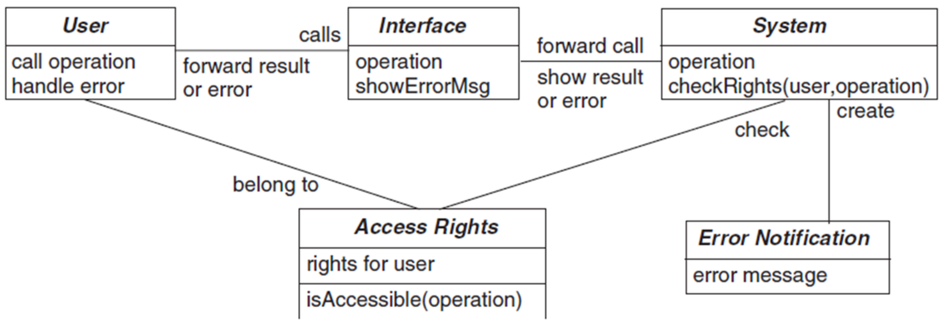


Figure 99: UML class diagram: Full View With Error Pattern [9]

FullViewWithError Pattern

$AccessRightCheck, ErrorMessage, Interface, systemComponent : \mathbb{CLASS}$
$check, checkRights : \mathbb{SIGNATURE}$
$Operations, TargetedOperations : \mathcal{P}\,\mathbb{SIGNATURE}$

$Call(checkRights \otimes systemComponent, check \otimes AccessRightCheck)$
$\text{TOTAL}(Call, TargetedOperations \otimes systemComponent, checkRights \otimes systemComponent)$
$\text{TOTAL}(Forward, Operations \otimes Interface, TargetedOperations \otimes systemComponent)$
$Produce(checkRights \otimes systemComponent, ErrorMessage)$

Figure 100: LePUS3 and Class-Z schema of Full View with Error Pattern

APPENDIX B: More Manual Conformance Checking Case Studies

Check Point pattern in JAAS

Implementation: JAAS

Claim in: [9]

Formalized pattern: Figure 31

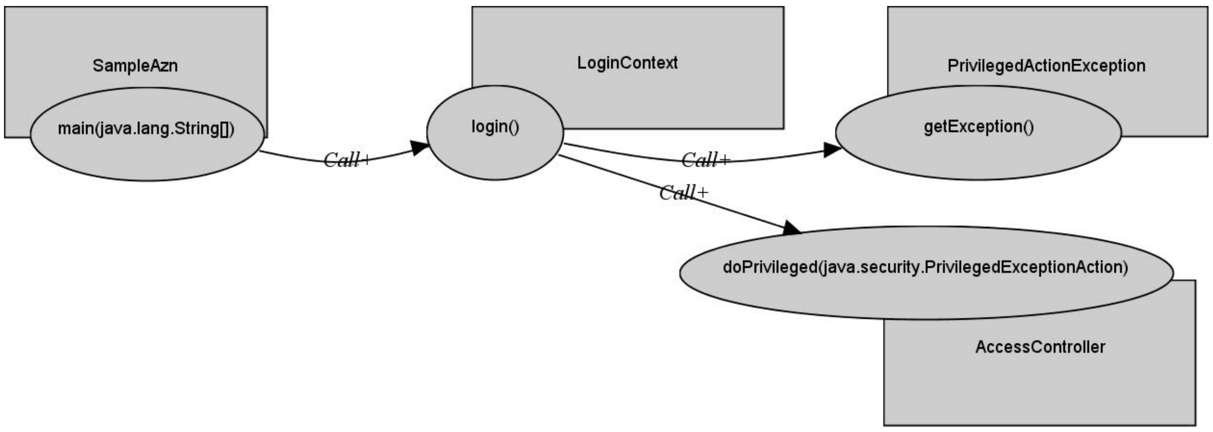


Figure 101: Codechart: An instance of Check Point in JAAS

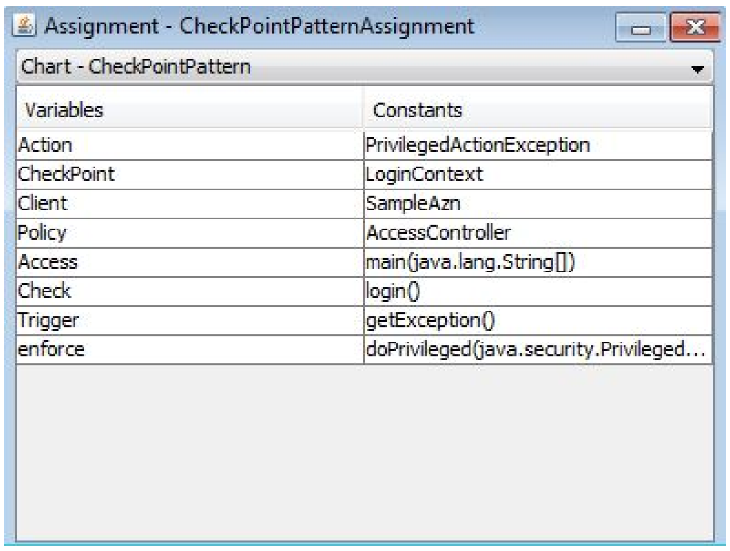
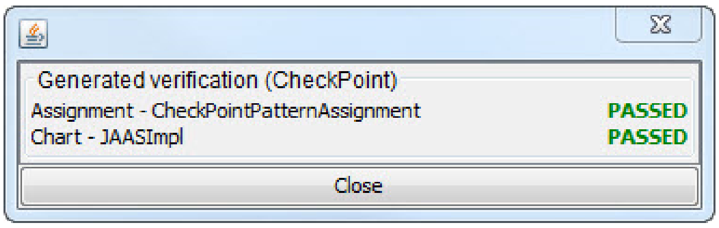

Assignment - CheckPointPatternAssignment

Chart - CheckPointPattern

Variables	Constants
Action	PrivilegedActionException
CheckPoint	LoginContext
Client	SampleAzn
Policy	AccessController
Access	main(java.lang.String[])
Check	login()
Trigger	getException()
enforce	doPrivileged(java.security.Privileged...

Figure 102: Manually created assignment of Check Point in JAAS


Generated verification (CheckPoint)

```
Assignment - CheckPointPatternAssignment          PASSED
Chart - JAASImpl                                  PASSED
```

Figure 103: Verification Result of Check Point instance in JAAS

Secure Architecture pattern

Implementation: JAAS

Claim in: [9]

Formalized pattern: Figure 89

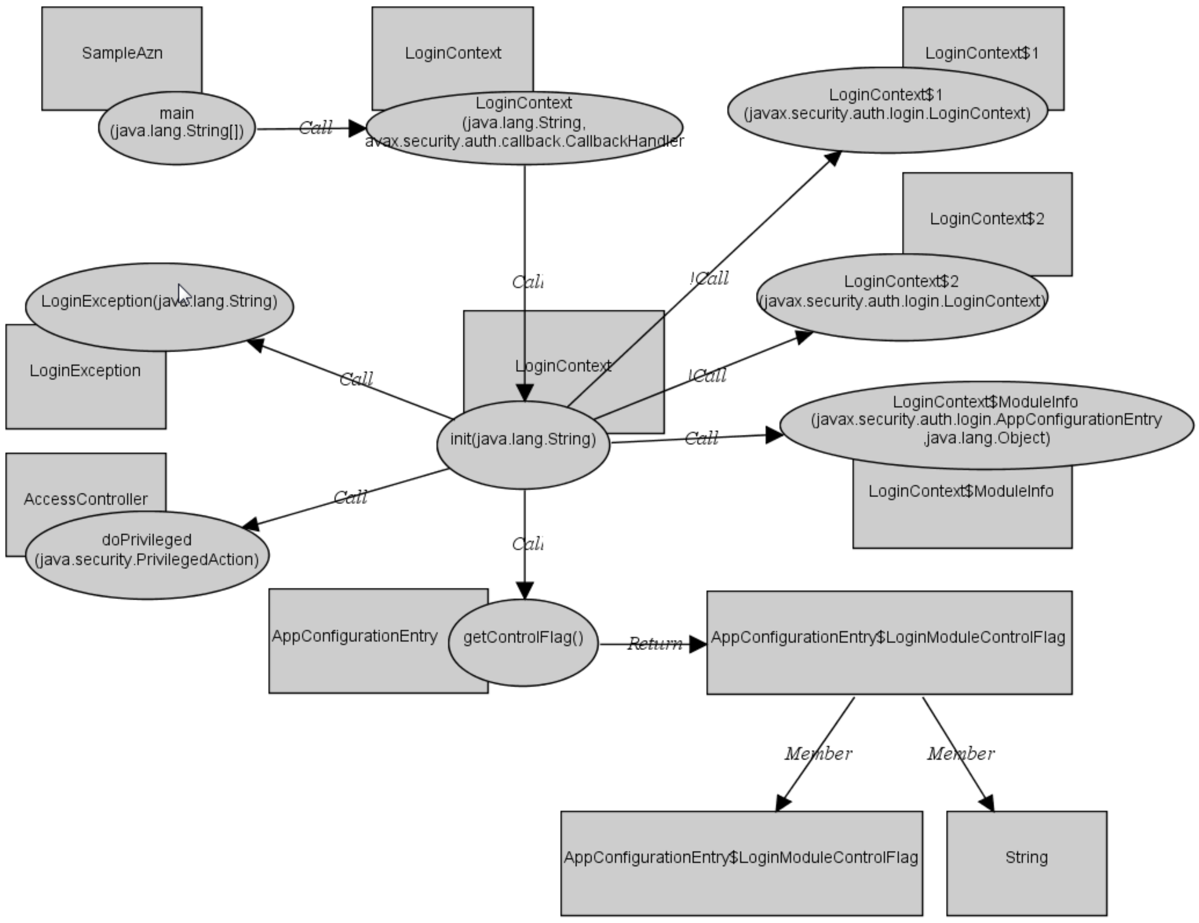

SampleAzn
main (java.lang.String[])
Call
LoginContext
LoginContext (java.lang.String, avax.security.auth.callback.CallbackHandler
LoginContext$1
LoginContext$1 (javax.security.auth.login.LoginContext)
LoginContext$2
LoginContext$2 (javax.security.auth.login.LoginContext)
!Call
!Call
LoginException(java.lang.String)
LoginException
Call
LoginContext
init(java.lang.String)
Call
LoginContext$ModuleInfo (javax.security.auth.login.AppConfigurationEntry ,java.lang.Object)
LoginContext$ModuleInfo
AccessController
doPrivileged (java.security.PrivilegedAction)
Call
Call
AppConfigurationEntry
getControlFlag()
Return
AppConfigurationEntry$LoginModuleControlFlag
Member
Member
AppConfigurationEntry$LoginModuleControlFlag
String

Figure 104: Codechart: An instance of Secure Architecture Pattern in JAAS

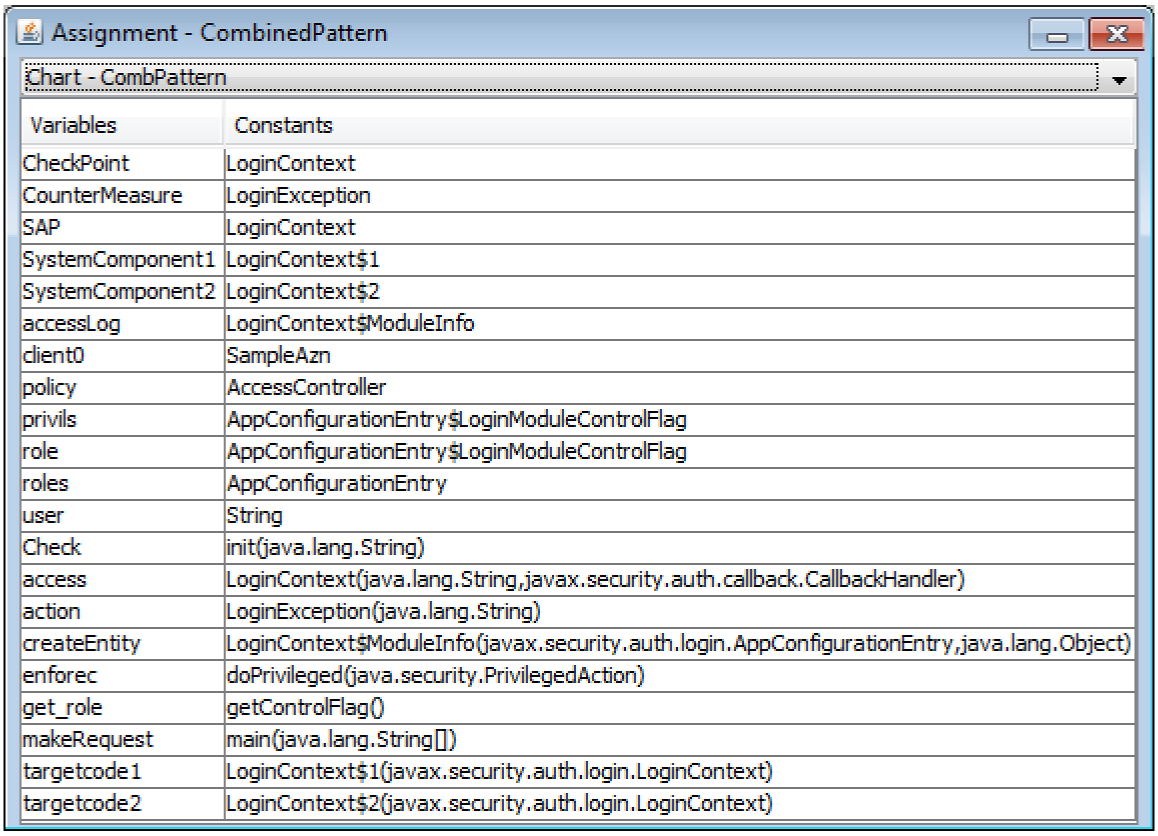
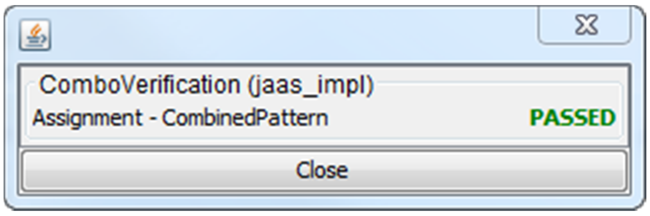

Assignment - CombinedPattern

Chart - CombPattern

Variables	Constants
CheckPoint	LoginContext
CounterMeasure	LoginException
SAP	LoginContext
SystemComponent1	LoginContext$1
SystemComponent2	LoginContext$2
accessLog	LoginContext$ModuleInfo
client0	SampleAzn
policy	AccessController
privils	AppConfigurationEntry$LoginModuleControlFlag
role	AppConfigurationEntry$LoginModuleControlFlag
roles	AppConfigurationEntry
user	String
Check	init(java.lang.String)
access	LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler)
action	LoginException(java.lang.String)
createEntity	LoginContext$ModuleInfo(javax.security.auth.login.AppConfigurationEntry,java.lang.Object)
enforec	doPrivileged(java.security.PrivilegedAction)
get_role	getControlFlag()
makeRequest	main(java.lang.String[])
targetcode1	LoginContext$1(javax.security.auth.login.LoginContext)
targetcode2	LoginContext$2(javax.security.auth.login.LoginContext)

Figure 105: Manually created assignment of Secure Architecture Pattern in JAAS


ComboVerification (jaas_impl)

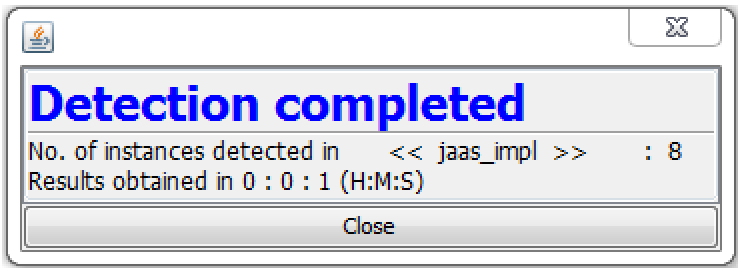
Assignment - CombinedPattern PASSED

Figure 106: Verification Result of Secure Architecture Pattern instance in JAAS

APPENDIX C: More Automatic Pattern Detection Case Studies

Check Point pattern in JAAS.

Formalized Security Pattern Figure 31



Detection completed

No. of instances detected in << jaas_impl >> : 8
Results obtained in 0 : 0 : 1 (H:M:S)

Figure 107: Results of detecting Check Point in JAAS

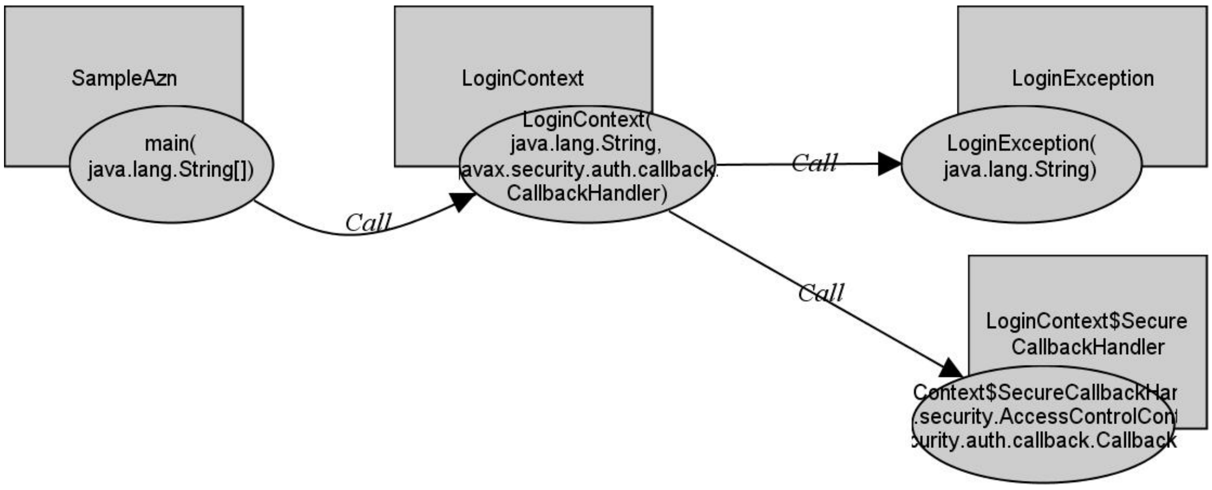
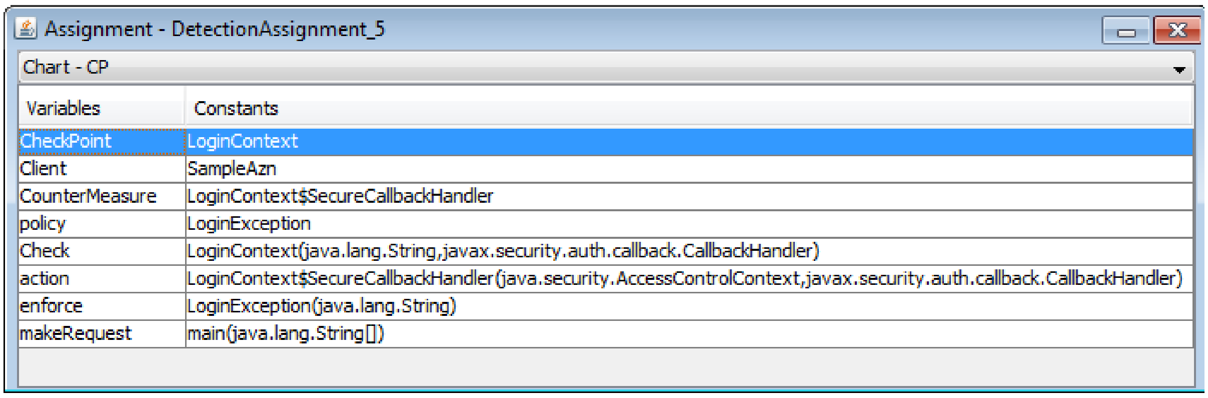

SampleAzn
main(java.lang.String[])
Call
LoginContext
LoginContext(java.lang.String, javax.security.auth.callback. CallbackHandler)
Call
LoginException
LoginException(java.lang.String)
Call
LoginContext$Secure CallbackHandler
Context$SecureCallbackHar .security.AccessControlCon curity.auth.callback.Callback

Figure 108: Codechart: An instance of Check Point detected in JAAS



Assignment - DetectionAssignment_5

Chart - CP

Variables	Constants
CheckPoint	LoginContext
Client	SampleAzn
CounterMeasure	LoginContext$SecureCallbackHandler
policy	LoginException
Check	LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler)
action	LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler)
enforce	LoginException(java.lang.String)
makeRequest	main(java.lang.String[])

Figure 109: an assignment of Check Point detected in in JAAS

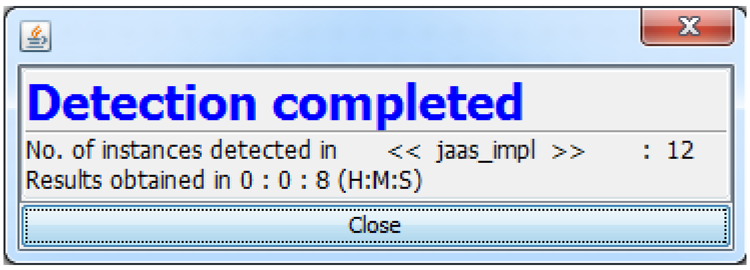
- **Results Explanation**

Client	SampleAzn
CheckPoint	LoginContext

CounterMeasure	LoginContext$SecureCallbackHandler, AccessController, LoginException, SecurityManager
Policy	LoginException, LoginContext$SecureCallbackHandler, AccessController, SecurityManager
Access	main(java.lang.String[])
Action	LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler), checkPermission(java.security.Permission), LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler), LoginException(java.lang.String), getContext(), doPrivileged(java.security.PrivilegedAction,java.security.AccessControlContext)
Check	LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler)
Enforce	LoginException(java.lang.String),checkPermission(java.security.Permission), doPrivileged(java.security.PrivilegedAction,java.security.AccessControlContext), getContext(), LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler)

Single Access Point & Check Point in JAAS.

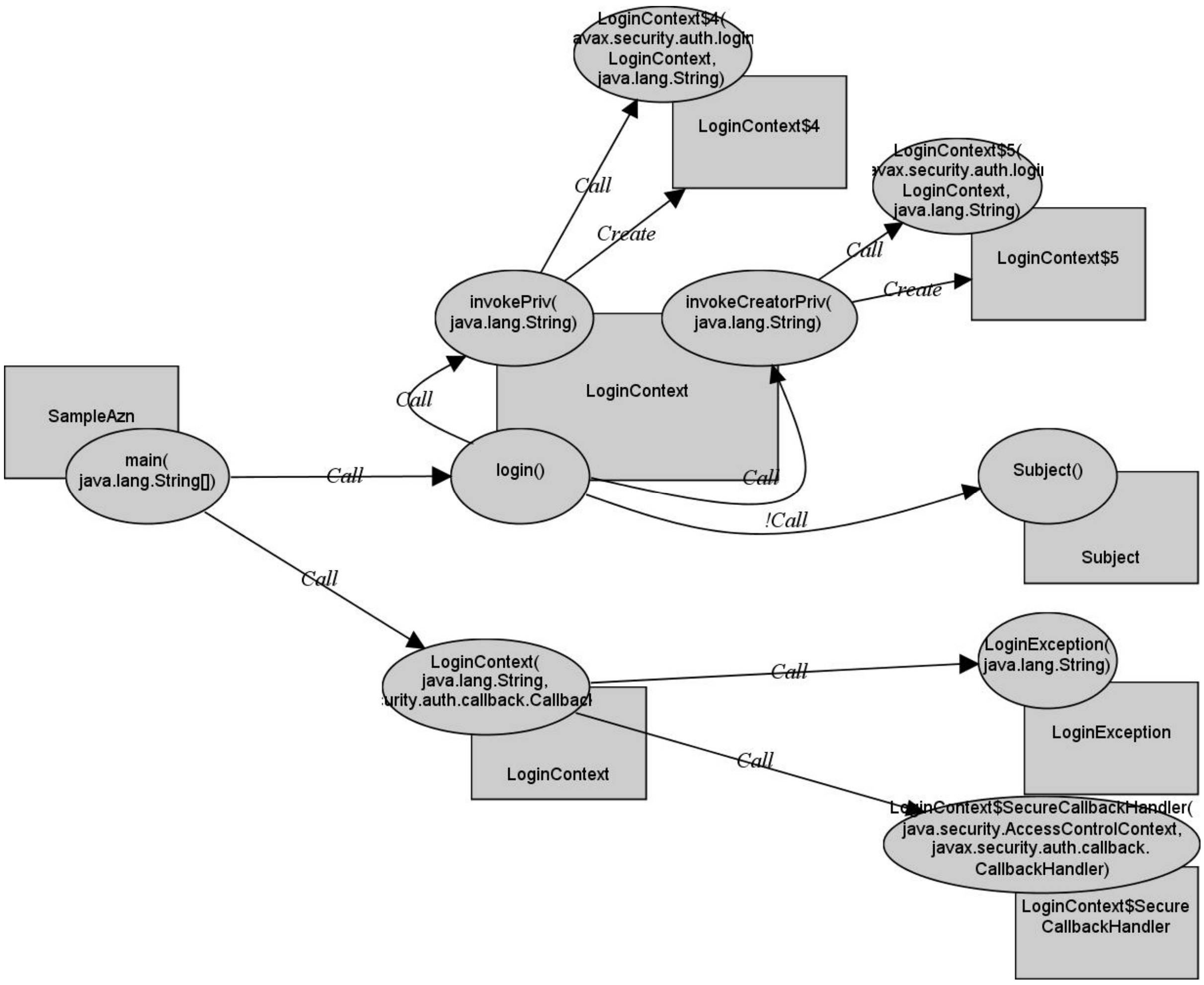
Formalized Security Pattern Figure 35


Detection completed

No. of instances detected in << jaas_impl >> : 12

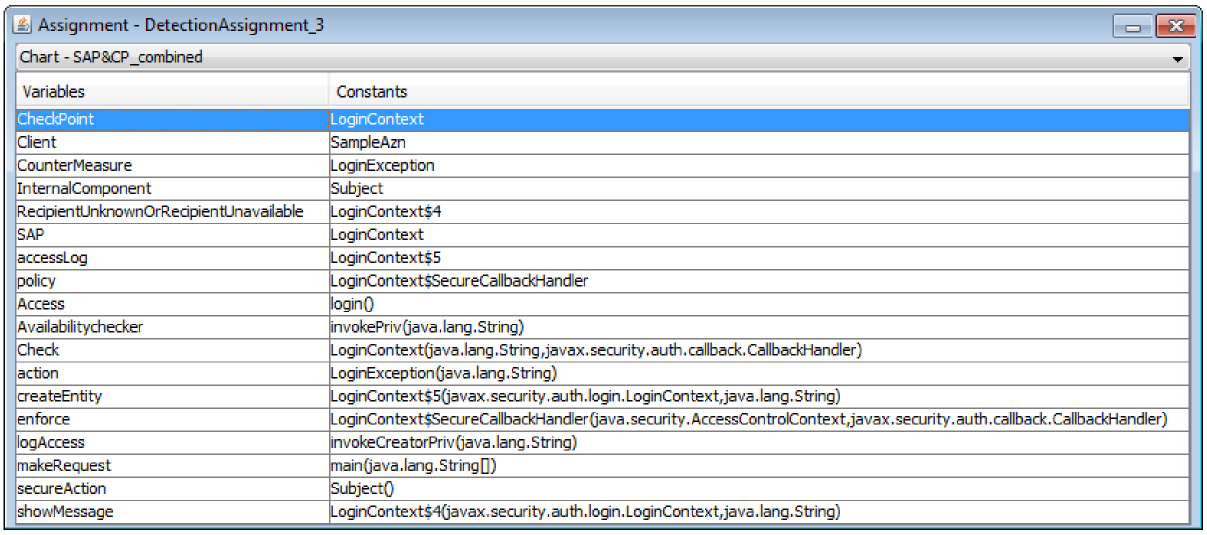
Results obtained in 0 : 0 : 8 (H:M:S)

Figure 110: Results of detecting Single Access Point & Check Point in JAAS



LoginContext$4(avax.security.auth.login LoginContext, java.lang.String)
LoginContext$4
Call
Create
LoginContext$5(avax.security.auth.logi LoginContext, java.lang.String)
LoginContext$5
Call
Create
invokePriv(java.lang.String)
invokeCreatorPriv(java.lang.String)
LoginContext
SampleAzn
main(java.lang.String[])
Call
Call
login()
Call
!Call
Subject()
Subject
Call
LoginContext(java.lang.String, urity.auth.callback.Callbac
LoginContext
Call
LoginException(java.lang.String)
LoginException
Call
LoginContext$SecureCallbackHandler(java.security.AccessControlContext, javax.security.auth.callback. CallbackHandler)
LoginContext$Secure CallbackHandler

Figure 111: Codechart: An instance of Single Access Point & Check Point detected in JAAS



Variables	Constants
CheckPoint	LoginContext
Client	SampleAzn
CounterMeasure	LoginException
InternalComponent	Subject
RecipientUnknownOrRecipientUnavailable	LoginContext$4
SAP	LoginContext
accessLog	LoginContext$5
policy	LoginContext$SecureCallbackHandler
Access	login()
Availabilitychecker	invokePriv(java.lang.String)
Check	LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler)
action	LoginException(java.lang.String)
createEntity	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)
enforce	LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler)
logAccess	invokeCreatorPriv(java.lang.String)
makeRequest	main(java.lang.String[])
secureAction	Subject()
showMessage	LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)

Figure 112: an assignment of Single Access Point & Check Point detected in JAAS

Results Explanation

Client	SampleAzn
CounterMeasure	{AccessController, LoginContext$SecureCallbackHandler, LoginException}
CheckPoint	LoginContext
SAP	LoginContext
Check	LoginContext(java.lang.String,javax.security.auth.callback.CallbackHandler)
Access	login()
InternalComponent	Subject
makeRequest	main(java.lang.String[])
secureAction	Subject()
RecipientUnknownOrRecipientUnavailable	{LoginContext$4, LoginContext$5}
accessLog	{LoginContext$4, LoginContext$5}
Policy	{LoginContext$SecureCallbackHandler, AccessController, LoginException}
Availabilitychecker	{invokePriv(java.lang.String), invokeCreatorPriv(java.lang.String)}
Action	{LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler), getContext(), LoginException(java.lang.String) }
createEntity	{LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String), LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String)}
Enforce	{LoginContext$SecureCallbackHandler(java.security.AccessControlContext,javax.security.auth.callback.CallbackHandler), LoginException(java.lang.String), getContext()}
logAccess	{invokePriv(java.lang.String), invokeCreatorPriv(java.lang.String)}
showMessage	{LoginContext$4(javax.security.auth.login.LoginContext,java.lang.String),

	LoginContext$5(javax.security.auth.login.LoginContext,java.lang.String)}

Full View with Error pattern in LEXI

Formalized Security Pattern Figure 98

Claim of implementation in LEXI is in [9]

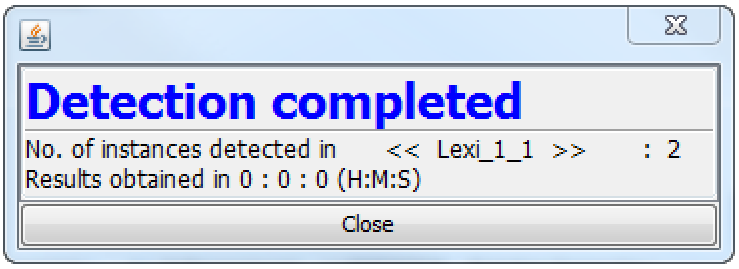

Detection completed
No. of instances detected in << Lexi_1_1 >> : 2
Results obtained in 0 : 0 : 0 (H:M:S)

Figure 113: Results of detecting Full View with Error pattern in LEXI

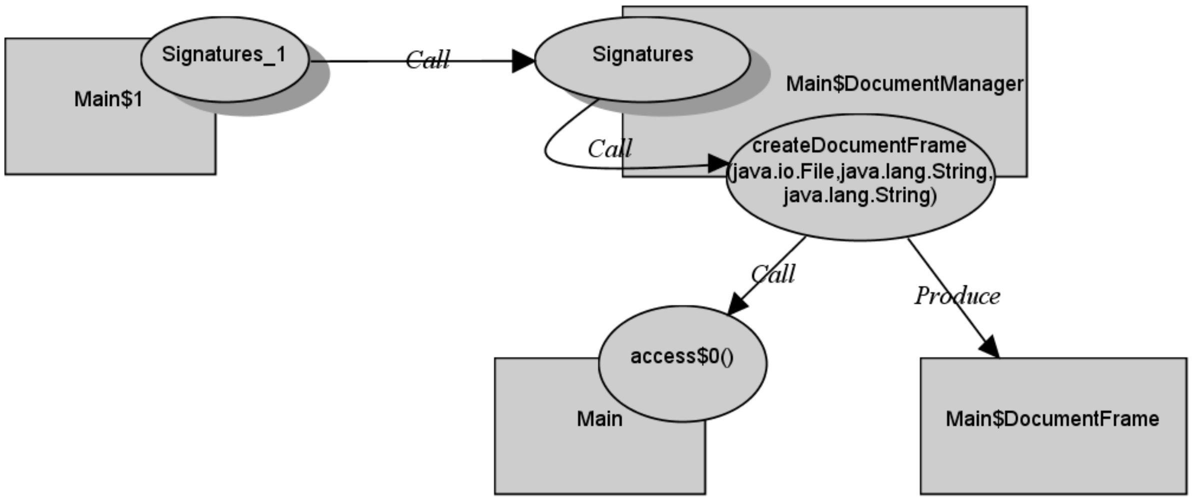
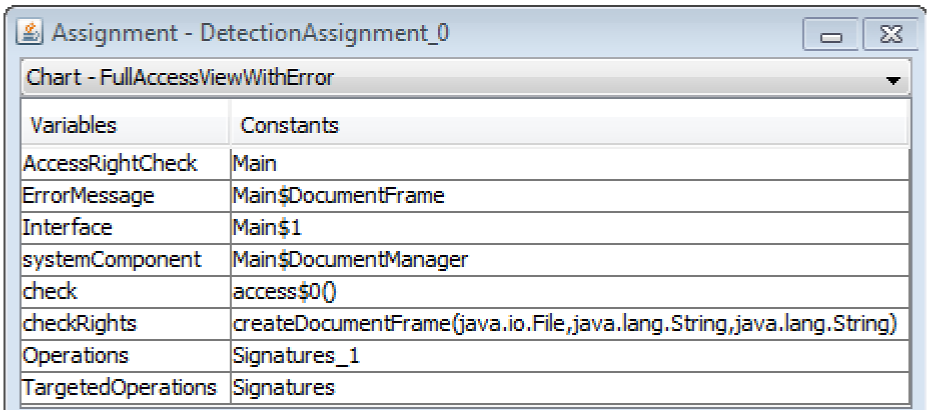


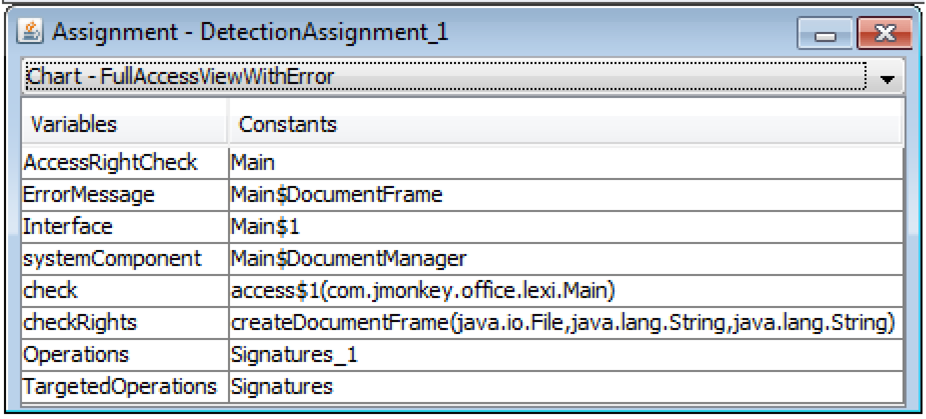
Figure 114: Codechart: An instance of Full View with Error pattern detected in LEXI



Assignment - DetectionAssignment_0

Chart - FullAccessViewWithError

Variables	Constants
AccessRightCheck	Main
ErrorMessage	Main$DocumentFrame
Interface	Main$1
systemComponent	Main$DocumentManager
check	access$0()
checkRights	createDocumentFrame(java.io.File,java.lang.String,java.lang.String)
Operations	Signatures_1
TargetedOperations	Signatures

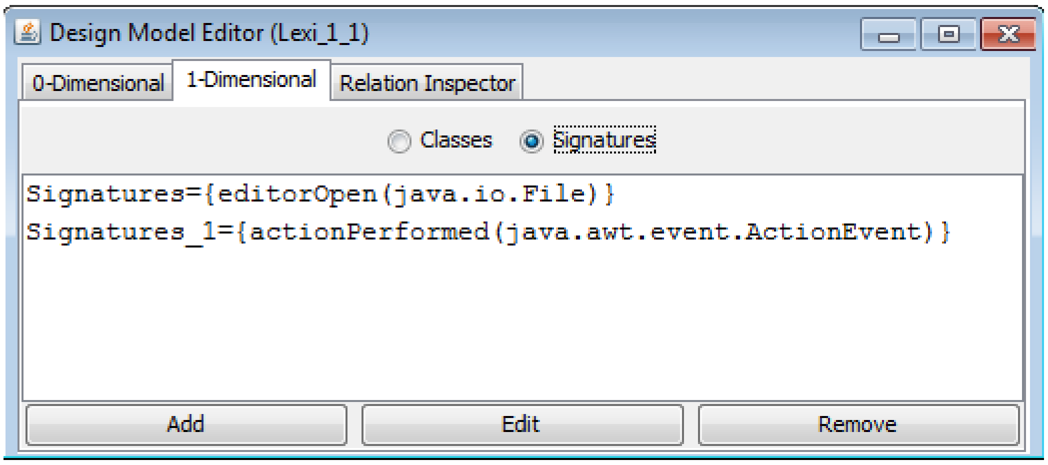


Assignment - DetectionAssignment_1

Chart - FullAccessViewWithError

Variables	Constants
AccessRightCheck	Main
ErrorMessage	Main$DocumentFrame
Interface	Main$1
systemComponent	Main$DocumentManager
check	access$1(com.jmonkey.office.lexi.Main)
checkRights	createDocumentFrame(java.io.File,java.lang.String,java.lang.String)
Operations	Signatures_1
TargetedOperations	Signatures

Figure 115: Assignments of Full View with Error pattern detected in LEXI



```
Signatures={editorOpen(java.io.File)}
Signatures_1={actionPerformed(java.awt.event.ActionEvent)}
```

Figure 116: Auto-defined higher-dimension entities (Methods) (Full View with Error pattern Validator - LEXI)

Class Adapter pattern Case Studies

Class Adapter pattern in JRefactory

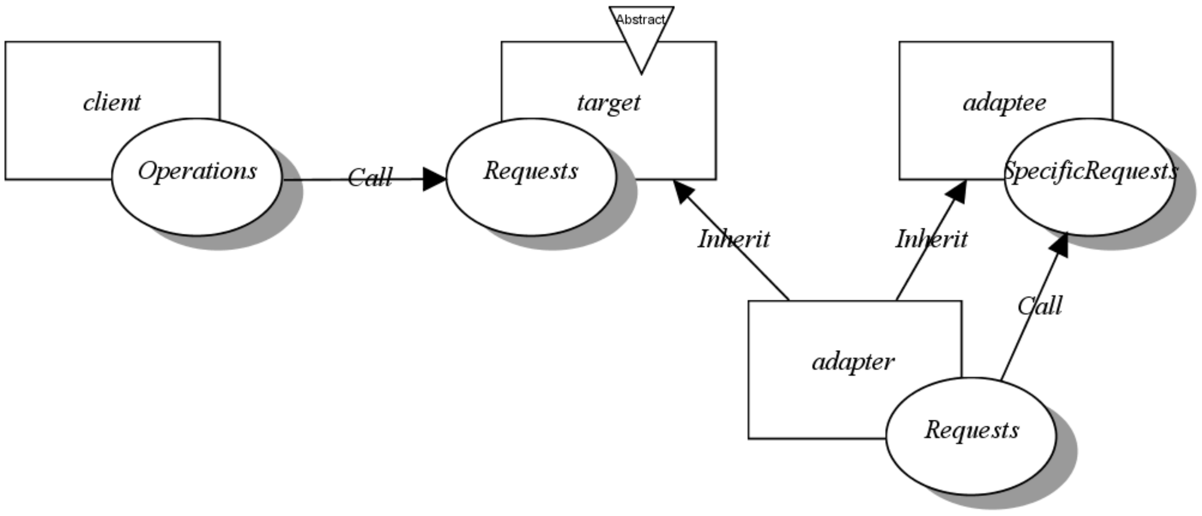
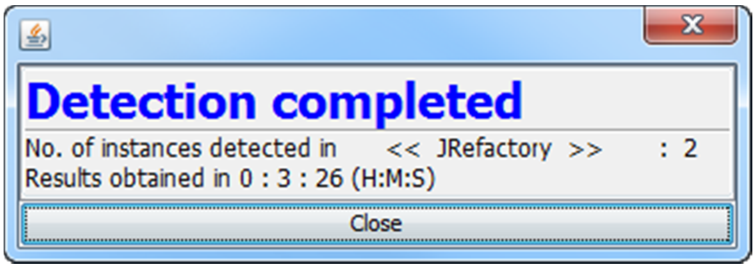


Figure 117: Codechart: Class Adapter pattern [38]



```
Detection completed
No. of instances detected in      << JRefactory >>      : 2
Results obtained in 0 : 3 : 26 (H:M:S)
```

Figure 118: Results of detecting Class Adapter in JRefactory

- **Results Explanation**

The difference is only with the client class. The rest are the same:

adaptee	LinedPanel
adapter	UMLPackage
client	{<**Symbol**:SaveMenuSelection,**FullQualifier**: net.sourceforge.jrefactory.uml.SaveMenuSelection>, <**Symbol**:org.acm.seguin.uml.SaveMenuSelection **FullQualifier**:org.acm.seguin.uml.SaveMenuSelection>}

target	Saveable
Operations	Signatures_3
Requests	Signatures_16
SpecificRequests	Signatures_22

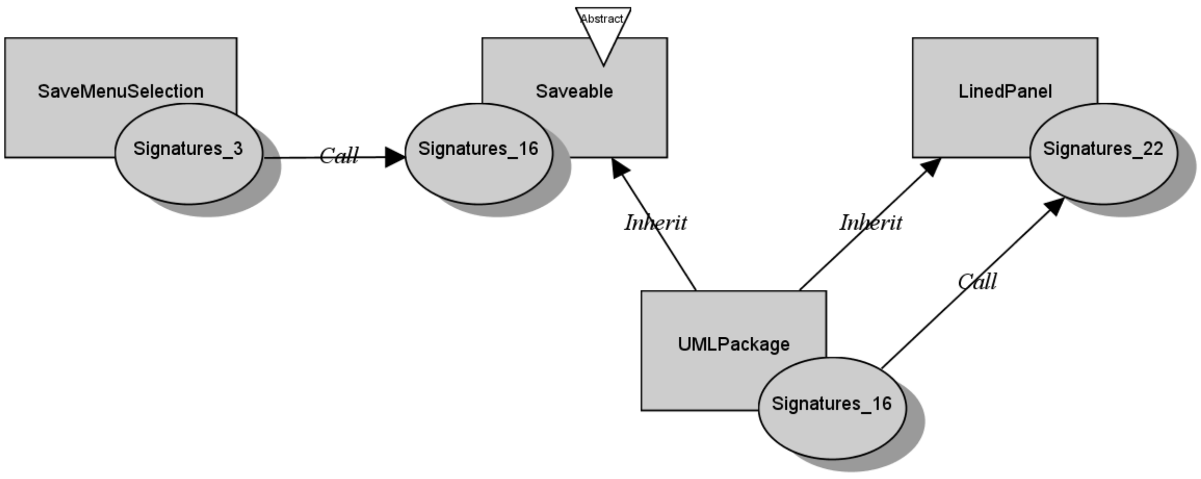


Figure 119: Codechart: An instance of Class Adapter pattern detected in JRefactory

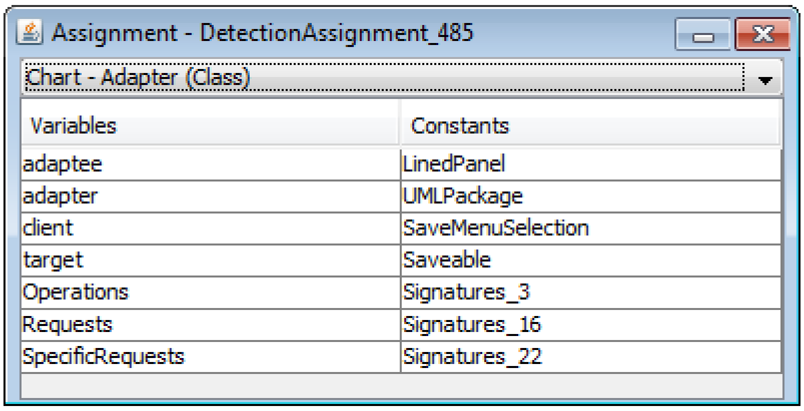
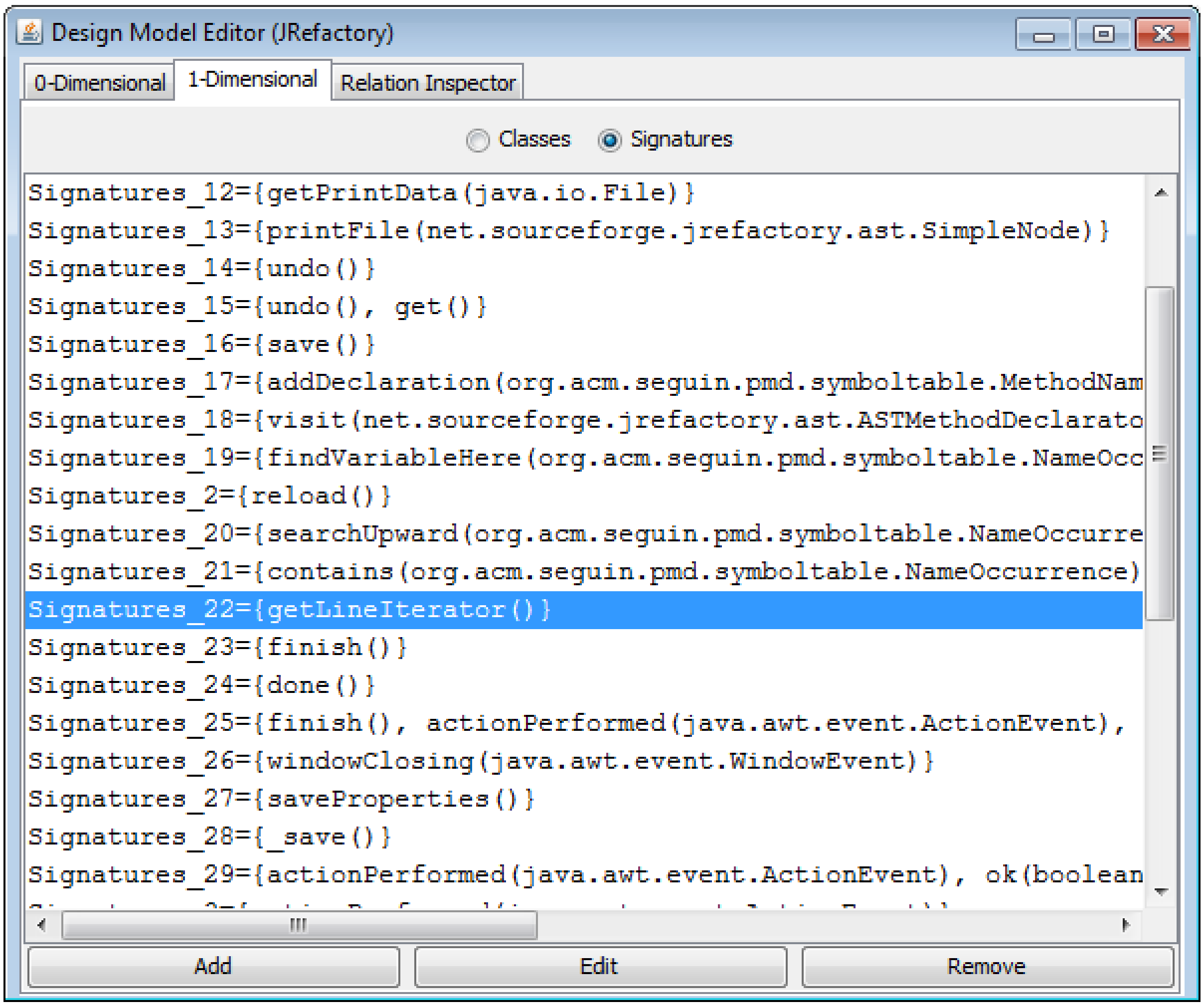

Assignment - DetectionAssignment_485

Chart - Adapter (Class)

Variables	Constants
adaptee	LinedPanel
adapter	UMLPackage
client	SaveMenuSelection
target	Saveable
Operations	Signatures_3
Requests	Signatures_16
SpecificRequests	Signatures_22

Figure 120: an assignment of Class Adapter detected in JRefactory



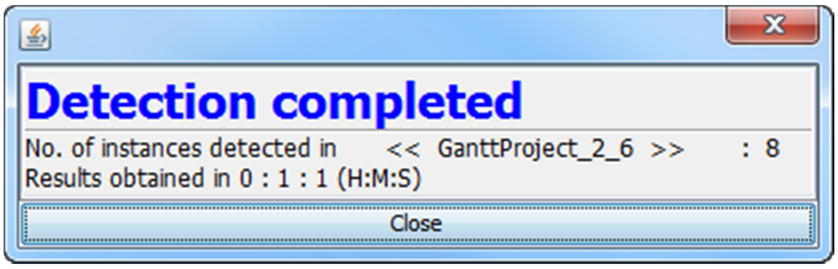
```
Signatures_12={getPrintData(java.io.File)}
Signatures_13={printFile(net.sourceforge.jrefactory.ast.SimpleNode)}
Signatures_14={undo()}
Signatures_15={undo(), get()}
Signatures_16={save()}
Signatures_17={addDeclaration(org.acm.seguin.pmd.symboltable.MethodNam
Signatures_18={visit(net.sourceforge.jrefactory.ast.ASTMethodDeclarato
Signatures_19={findVariableHere(org.acm.seguin.pmd.symboltable.NameOcc
Signatures_2={reload()}
Signatures_20={searchUpward(org.acm.seguin.pmd.symboltable.NameOccurre
Signatures_21={contains(org.acm.seguin.pmd.symboltable.NameOccurrence)
Signatures_22={getLineIterator()}
Signatures_23={finish()}
Signatures_24={done()}
Signatures_25={finish(), actionPerformed(java.awt.event.ActionEvent),
Signatures_26={windowClosing(java.awt.event.WindowEvent)}
Signatures_27={saveProperties()}
Signatures_28={_save()}
Signatures_29={actionPerformed(java.awt.event.ActionEvent), ok(boolean
```

Figure 121: Auto-defined higher-dimension entities (Methods) (Class Adapter - JRefactory)

Signatures Set details

Signatures_16={save()}
 	
Signatures_22={getLineIterator()}

Class Adapter pattern in GanttProject



Detection completed

```
No. of instances detected in    << GanttProject_2_6 >>    : 8
Results obtained in 0 : 1 : 1 (H:M:S)
```

Figure 122: Results of detecting Class Adapter in GanttProject

- **Results Explanation**

adaptee	{ConstraintImpl}
adapter	{FinishFinishConstraintImpl, FinishStartConstraintImpl, StartStartConstraintImpl, StartFinishConstraintImpl}
client	{RecalculateTaskScheduleAlgorithm, CriticalPathAlgorithmImpl$Processor}
target	{TaskDependencyConstraint}
Operations	{Signatures_26, Signatures_29}
Requests	{Signatures_25, Signatures_28}
SpecificRequests	{Signatures_24, Signatures_27}

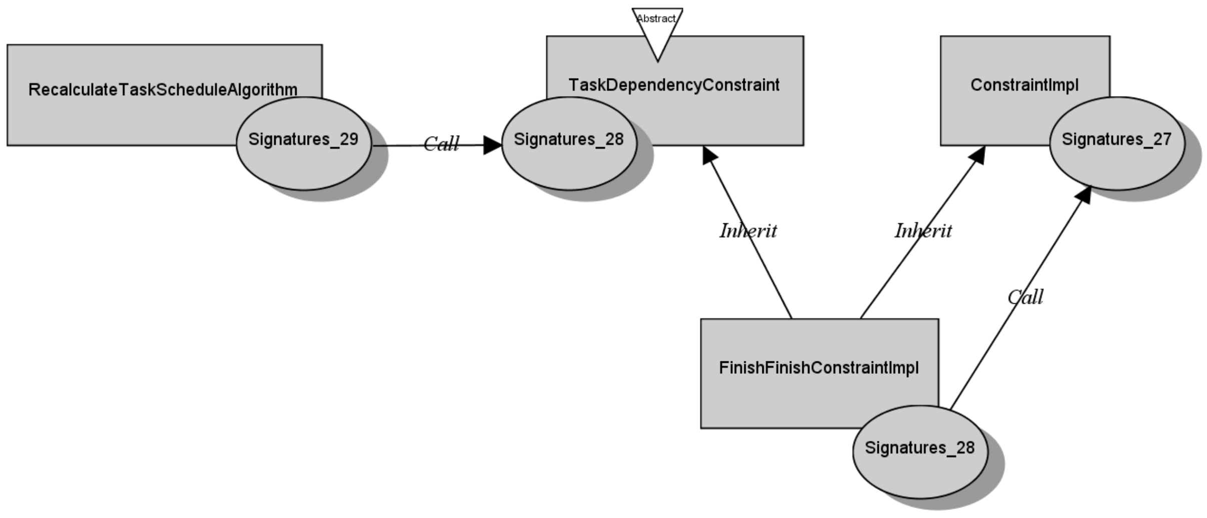
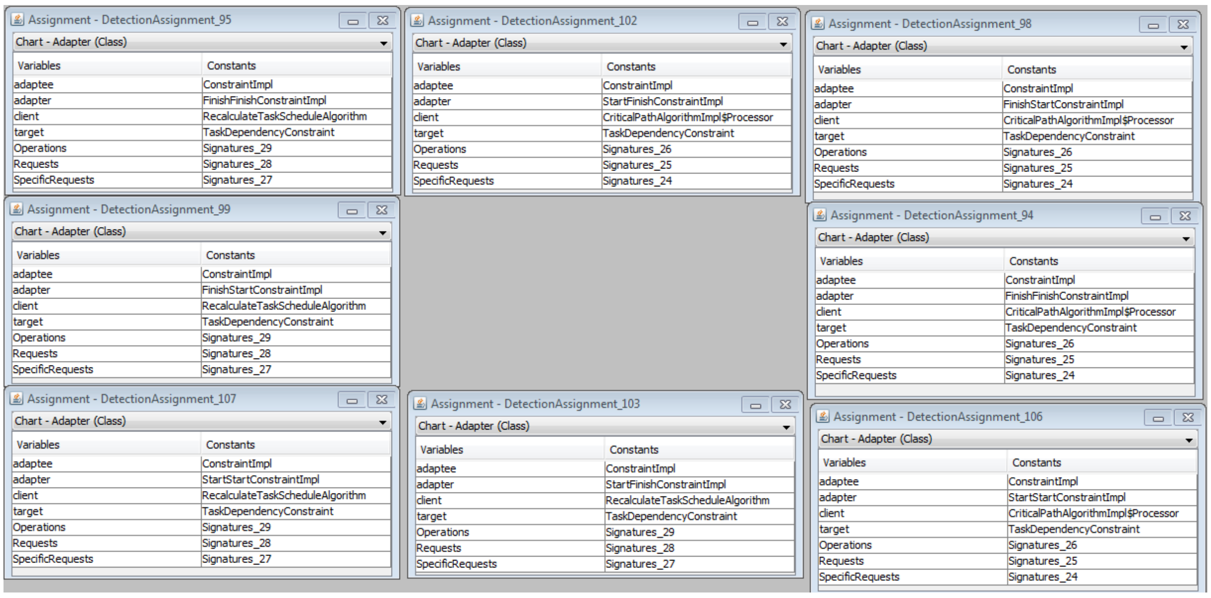


Figure 123: Codechart: An instance of Class Adapter pattern detected in GanttProject



Assignment - DetectionAssignment_95

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	FinishFinishConstraintImpl
client	RecalculateTaskScheduleAlgorithm
target	TaskDependencyConstraint
Operations	Signatures_29
Requests	Signatures_28
SpecificRequests	Signatures_27

Assignment - DetectionAssignment_102

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	StartFinishConstraintImpl
client	CriticalPathAlgorithmImpl$Processor
target	TaskDependencyConstraint
Operations	Signatures_26
Requests	Signatures_25
SpecificRequests	Signatures_24

Assignment - DetectionAssignment_98

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	FinishStartConstraintImpl
client	CriticalPathAlgorithmImpl$Processor
target	TaskDependencyConstraint
Operations	Signatures_26
Requests	Signatures_25
SpecificRequests	Signatures_24

Assignment - DetectionAssignment_99

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	FinishStartConstraintImpl
client	RecalculateTaskScheduleAlgorithm
target	TaskDependencyConstraint
Operations	Signatures_29
Requests	Signatures_28
SpecificRequests	Signatures_27

Assignment - DetectionAssignment_94

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	FinishFinishConstraintImpl
client	CriticalPathAlgorithmImpl$Processor
target	TaskDependencyConstraint
Operations	Signatures_26
Requests	Signatures_25
SpecificRequests	Signatures_24

Assignment - DetectionAssignment_107

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	StartStartConstraintImpl
client	RecalculateTaskScheduleAlgorithm
target	TaskDependencyConstraint
Operations	Signatures_29
Requests	Signatures_28
SpecificRequests	Signatures_27

Assignment - DetectionAssignment_103

Chart - Adapter (Class)

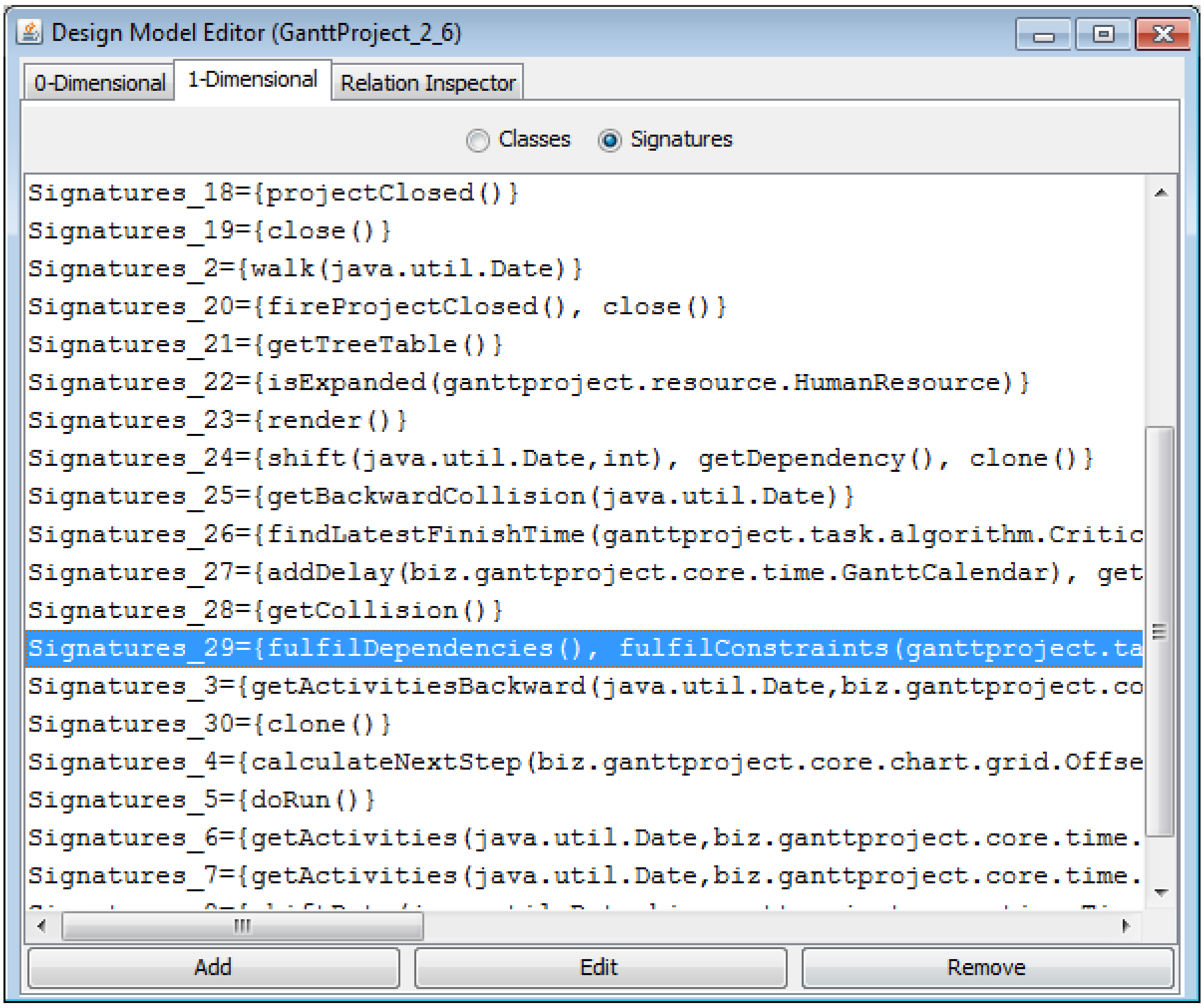
Variables	Constants
adaptee	ConstraintImpl
adapter	StartFinishConstraintImpl
client	RecalculateTaskScheduleAlgorithm
target	TaskDependencyConstraint
Operations	Signatures_29
Requests	Signatures_28
SpecificRequests	Signatures_27

Assignment - DetectionAssignment_106

Chart - Adapter (Class)

Variables	Constants
adaptee	ConstraintImpl
adapter	StartStartConstraintImpl
client	CriticalPathAlgorithmImpl$Processor
target	TaskDependencyConstraint
Operations	Signatures_26
Requests	Signatures_25
SpecificRequests	Signatures_24

Figure 124: Assignments of Class Adapter detected in GanttProject



Design Model Editor (GanttProject_2_6)

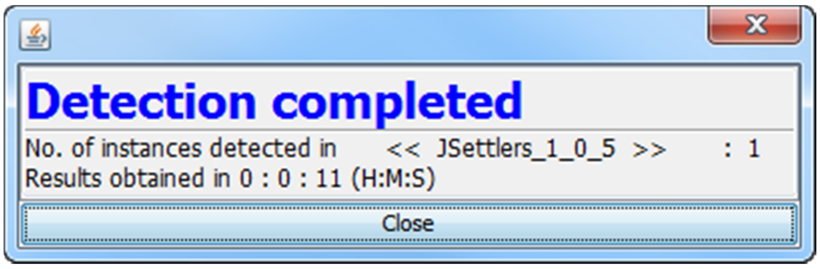
```
Signatures_18={projectClosed()}
Signatures_19={close()}
Signatures_2={walk(java.util.Date)}
Signatures_20={fireProjectClosed(), close()}
Signatures_21={getTreeTable()}
Signatures_22={isExpanded(ganttproject.resource.HumanResource)}
Signatures_23={render()}
Signatures_24={shift(java.util.Date,int), getDependency(), clone()}
Signatures_25={getBackwardCollision(java.util.Date)}
Signatures_26={findLatestFinishTime(ganttproject.task.algorithm.Critic
Signatures_27={addDelay(biz.ganttproject.core.time.GanttCalendar), get
Signatures_28={getCollision()}
Signatures_29={fulfilDependencies(), fulfilConstraints(ganttproject.ta
Signatures_3={getActivitiesBackward(java.util.Date,biz.ganttproject.co
Signatures_30={clone()}
Signatures_4={calculateNextStep(biz.ganttproject.core.chart.grid.Offse
Signatures_5={doRun()}
Signatures_6={getActivities(java.util.Date,biz.ganttproject.core.time.
Signatures_7={getActivities(java.util.Date,biz.ganttproject.core.time.
```

Figure 125: Auto-defined higher-dimension entities (Methods) (Class Adapter - GanttProject)

Signatures Set details

Signatures_24={shift(java.util.Date,int), getDependency(), clone()}
 	
Signatures_25={getBackwardCollision(java.util.Date)}
 	
Signatures_26={findLatestFinishTime(ganttproject.task.algorithm.CriticalPathAlgorithmImpl$Node,ganttproject.task.algorithm.CriticalPathAlgorithmImpl$Node,ganttproject.task.dependency.TaskDependency)}
 	
Signatures_27={addDelay(biz.ganttproject.core.time.GanttCalendar), getDependency(), clone()}
 	
Signatures_28={getCollision()}
 	
Signatures_29={fulfilDependencies(), fulfilConstraints(ganttproject.task.dependency.TaskDependency)}

Class Adapter pattern in JSettlers


Detection completed

No. of instances detected in << JSettlers_1_0_5 >> : 1

Results obtained in 0 : 0 : 11 (H:M:S)

Figure 126: Results of detecting Class Adapter in JSettlers

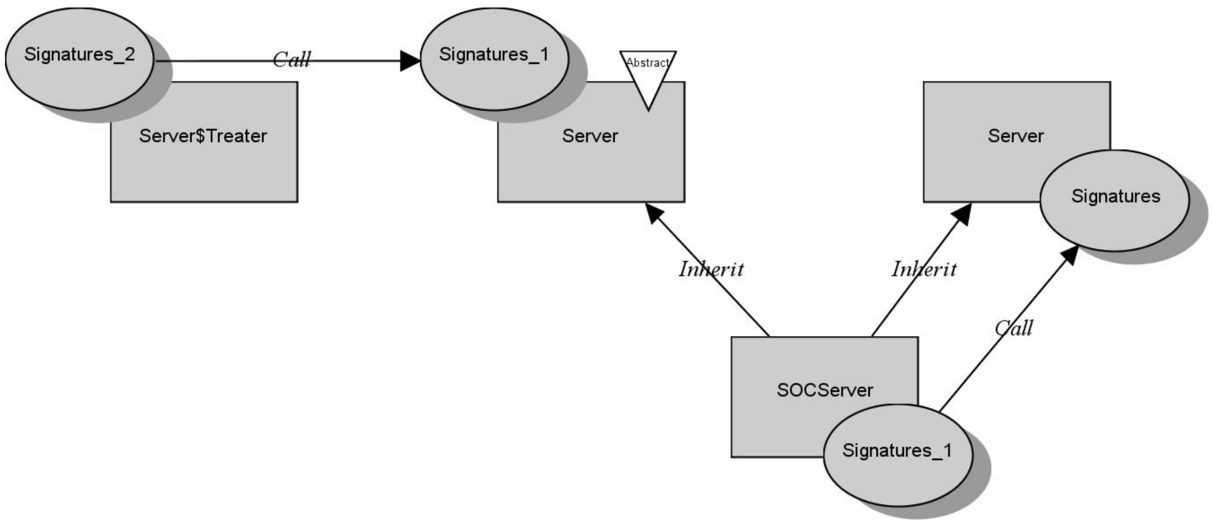


Figure 127: Codechart: An instance of Class Adapter pattern detected in JSettlers

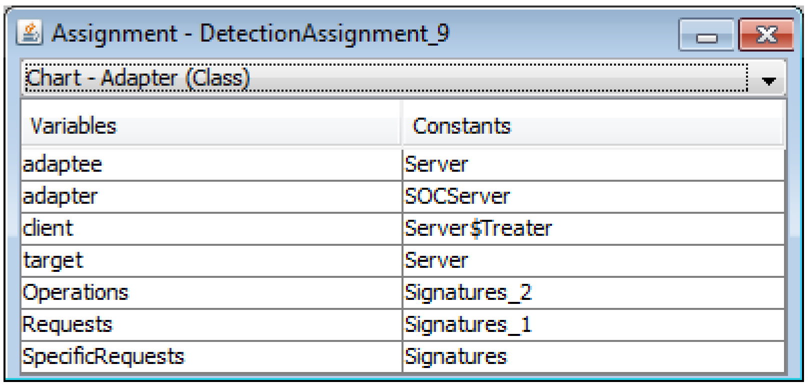
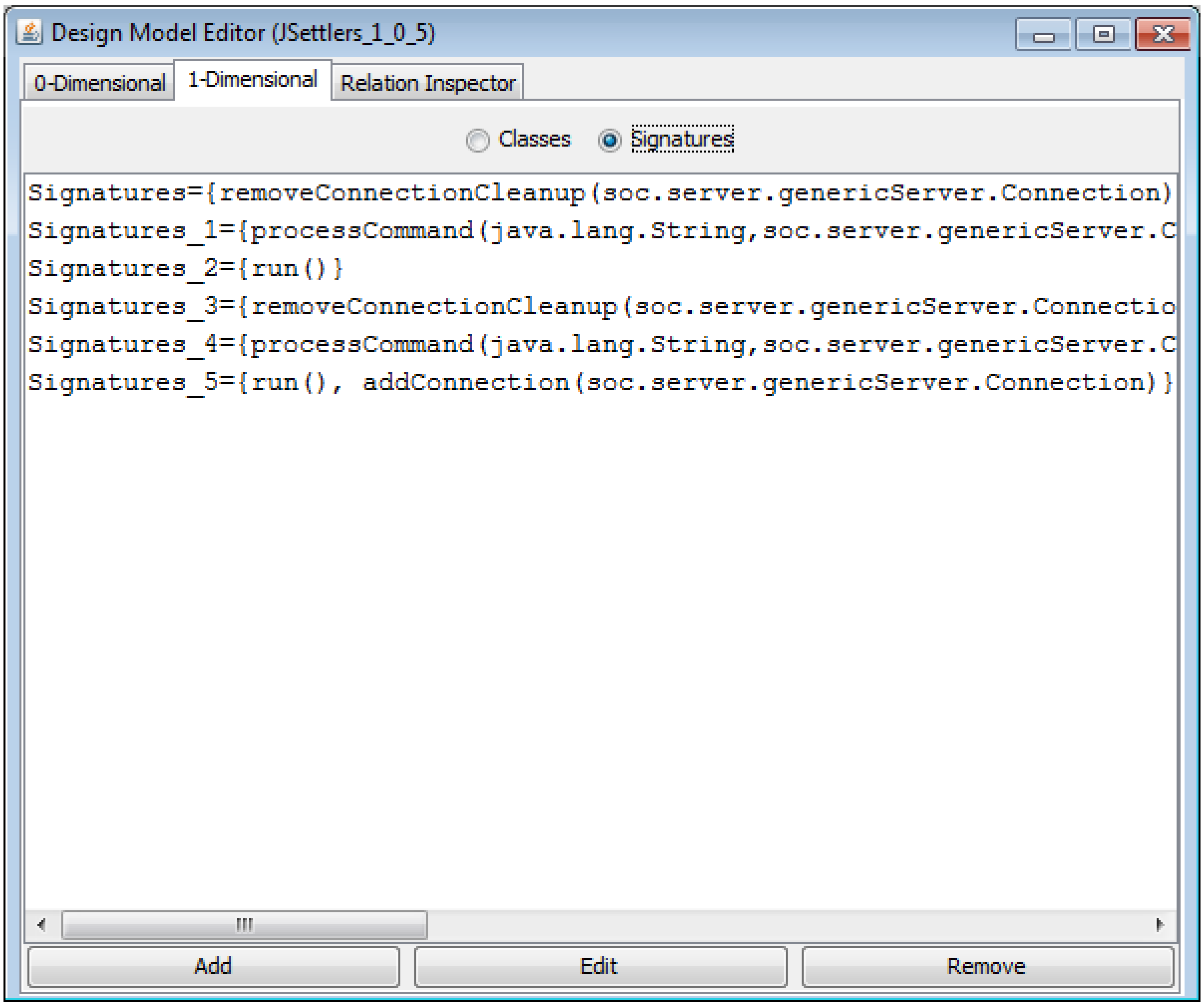

Assignment - DetectionAssignment_9

Chart - Adapter (Class)

Variables	Constants
adaptee	Server
adapter	SOCServer
client	Server$Treater
target	Server
Operations	Signatures_2
Requests	Signatures_1
SpecificRequests	Signatures

Figure 128: An assignment of Class Adapter detected in JSettlers



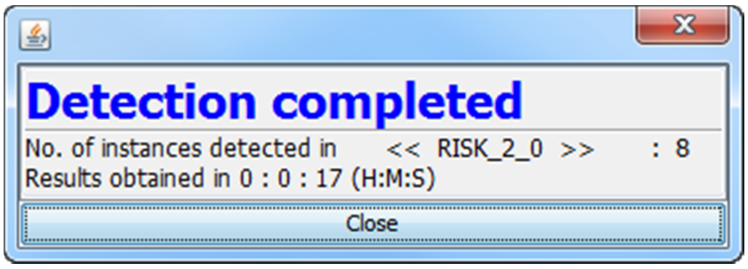
```
Signatures={removeConnectionCleanup(soc.server.genericServer.Connection)
Signatures_1={processCommand(java.lang.String,soc.server.genericServer.C
Signatures_2={run()}
Signatures_3={removeConnectionCleanup(soc.server.genericServer.Connectio
Signatures_4={processCommand(java.lang.String,soc.server.genericServer.C
Signatures_5={run(), addConnection(soc.server.genericServer.Connection)}
```

Figure 129: Auto-defined higher-dimension entities (Methods) (Class Adapter - JSettlers)

Signatures Set details

Signatures={removeConnectionCleanup(soc.server.genericServer.Connection), removeConnection(soc.server.genericServer.Connection), connectionCount(), stopServer(), broadcast(java.lang.String)}
 	
Signatures_1={processCommand(java.lang.String,soc.server.genericServer.Connection)}
 	
Signatures_2={run()}

Class Adapter pattern in RISK 2.0



Detection completed

```
No. of instances detected in    << RISK_2_0 >>    : 8
Results obtained in 0 : 0 : 17 (H:M:S)
```

Figure 130: Results of detecting Class Adapter in RISK 2.0

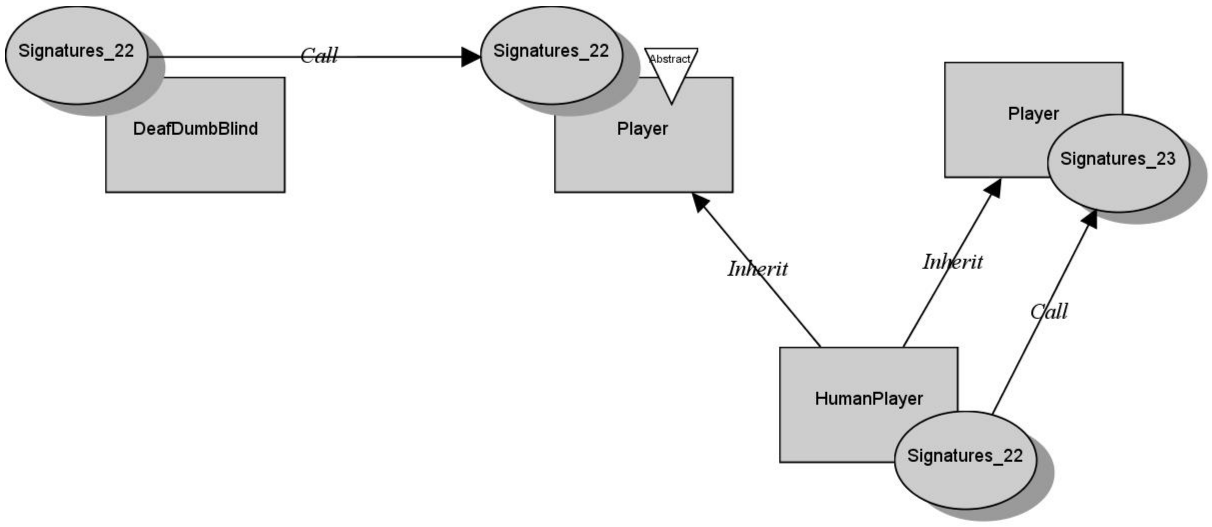
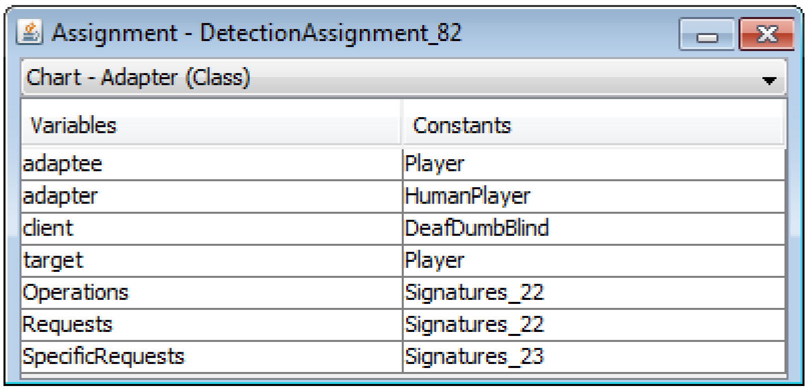


Figure 131: Codechart: An instance of Class Adapter pattern detected in RISK 2.0

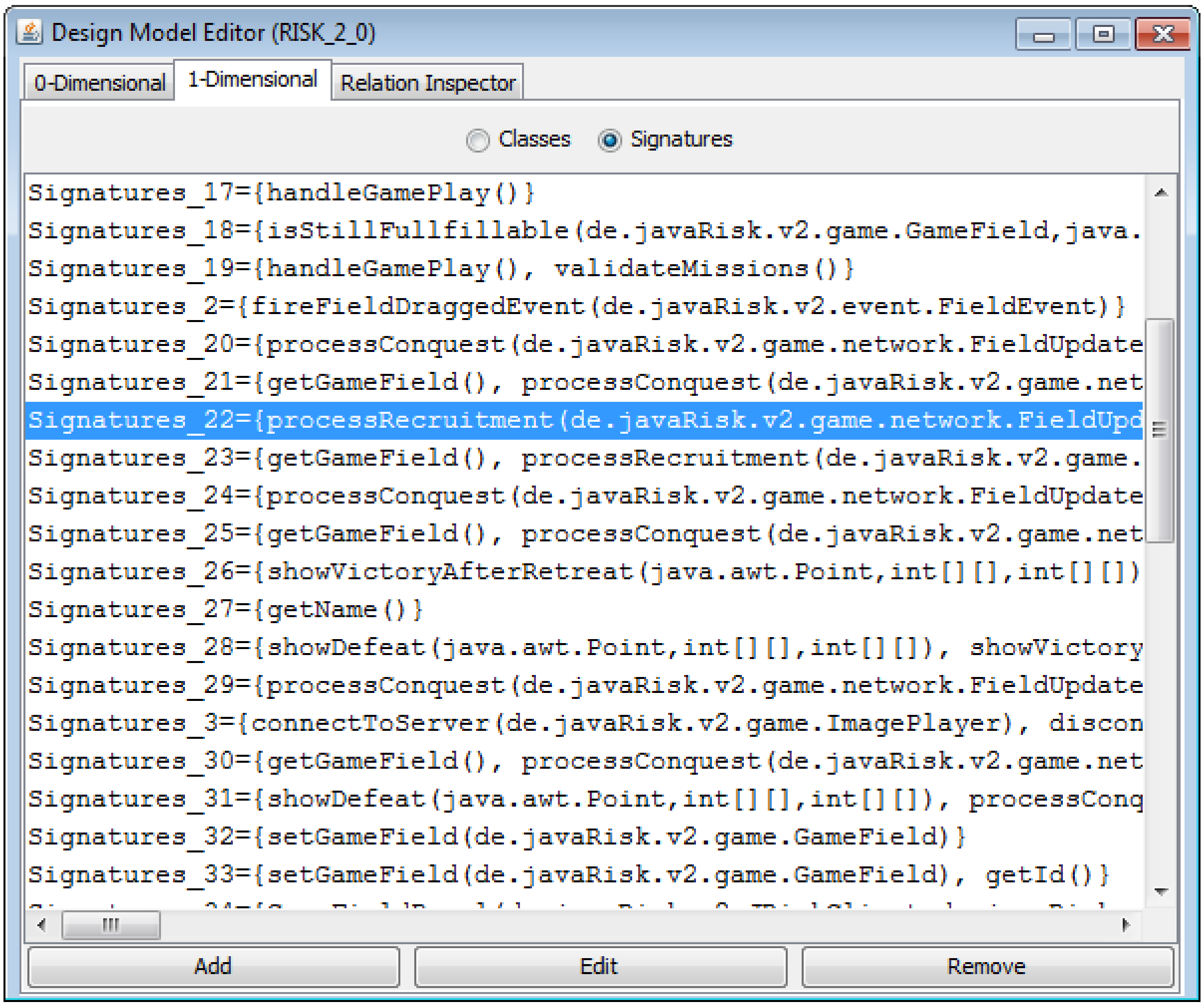


Assignment - DetectionAssignment_82

Chart - Adapter (Class)

Variables	Constants
adaptee	Player
adapter	HumanPlayer
client	DeafDumbBlind
target	Player
Operations	Signatures_22
Requests	Signatures_22
SpecificRequests	Signatures_23

Figure 132: An assignment of Class Adapter detected in RISK 2.0



```
Signatures_17={handleGamePlay()}
Signatures_18={isStillFullfillable(de.javaRisk.v2.game.GameField,java.
Signatures_19={handleGamePlay(), validateMissions()}
Signatures_2={fireFieldDraggedEvent(de.javaRisk.v2.event.FieldEvent)}
Signatures_20={processConquest(de.javaRisk.v2.game.network.FieldUpdate
Signatures_21={getGameField(), processConquest(de.javaRisk.v2.game.net
Signatures_22={processRecruitment(de.javaRisk.v2.game.network.FieldUpd
Signatures_23={getGameField(), processRecruitment(de.javaRisk.v2.game.
Signatures_24={processConquest(de.javaRisk.v2.game.network.FieldUpdate
Signatures_25={getGameField(), processConquest(de.javaRisk.v2.game.net
Signatures_26={showVictoryAfterRetreat(java.awt.Point,int[][],int[][])
Signatures_27={getName()}
Signatures_28={showDefeat(java.awt.Point,int[][],int[][]), showVictory
Signatures_29={processConquest(de.javaRisk.v2.game.network.FieldUpdate
Signatures_3={connectToServer(de.javaRisk.v2.game.ImagePlayer), discon
Signatures_30={getGameField(), processConquest(de.javaRisk.v2.game.net
Signatures_31={showDefeat(java.awt.Point,int[][],int[][]), processConq
Signatures_32={setGameField(de.javaRisk.v2.game.GameField)}
Signatures_33={setGameField(de.javaRisk.v2.game.GameField), getId()}
```

Figure 133: Auto-defined higher-dimension entities (Methods) (Class Adapter - RISK 2.0)

Signatures Set details

Signatures_22={processRecruitment(de.javaRisk.v2.game.network.FieldUpdate), processMovement(de.javaRisk.v2.game.ArmyCollection)}
 	
Signatures_23={getGameField(), processRecruitment(de.javaRisk.v2.game.network.FieldUpdate), processMovement(de.javaRisk.v2.game.ArmyCollection), isInTurn()}
 	
Signatures_24={processConquest(de.javaRisk.v2.game.network.FieldUpdate), processDestruction(de.javaRisk.v2.game.ArmyCollection), setGameField(de.javaRisk.v2.game.GameField), processBuild(de.javaRisk.v2.game.network.FieldUpdate), processPullDown(de.javaRisk.v2.game.network.FieldUpdate), processRecruitment(de.javaRisk.v2.game.network.FieldUpdate), processMovement(de.javaRisk.v2.game.ArmyCollection)}

Signatures_25={getGameField(), processConquest(de.javaRisk.v2.game.network.FieldUpdate), processDestruction(de.javaRisk.v2.game.ArmyCollection), setGameField(de.javaRisk.v2.game.GameField), processBuild(de.javaRisk.v2.game.network.FieldUpdate), processPullDown(de.javaRisk.v2.game.network.FieldUpdate), processRecruitment(de.javaRisk.v2.game.network.FieldUpdate), processMovement(de.javaRisk.v2.game.ArmyCollection), getId(), isInTurn()}

Signatures_32={setGameField(de.javaRisk.v2.game.GameField)}

Signatures_43={displayGameField(de.javaRisk.v2.game.GameField,java.lang.String)}

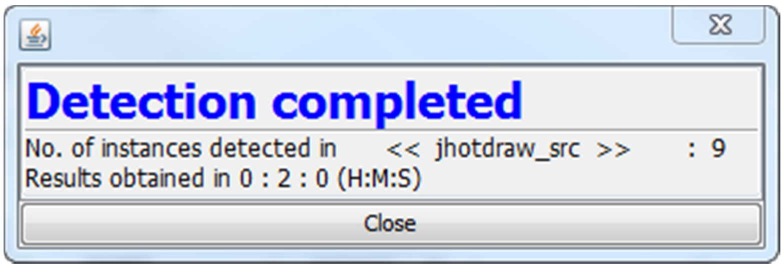
Signatures_44={drawGameField(de.javaRisk.v2.tools.DrawProperties)}

Signatures_45={getGameField(), getCallingComponent()}

Signatures_46={paintComponent(java.awt.Graphics)}

Signatures_47={setCenteredField(java.awt.Point), setGameField(de.javaRisk.v2.game.GameField)}

Class Adapter pattern in JHotdraw


Detection completed

```
No. of instances detected in     << jhotdraw_src >>     : 9
Results obtained in 0 : 2 : 0 (H:M:S)
```

Figure 134: Results of detecting Class Adapter in JHotdraw

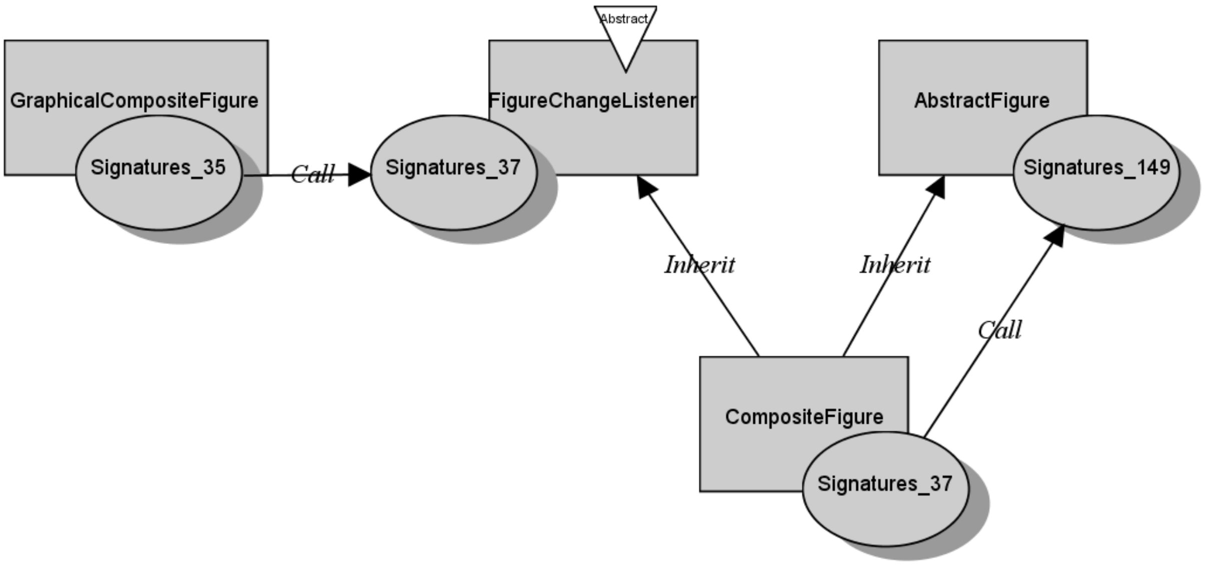
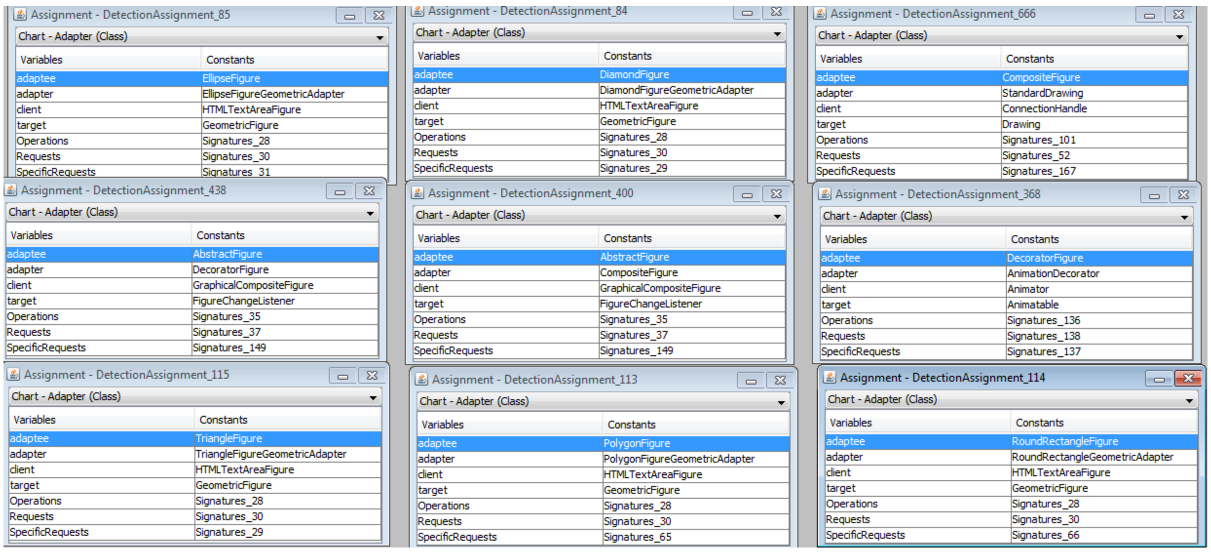


Figure 135: Codechart: An instance of Adapter pattern detected in JHotdraw



Assignment - DetectionAssignment_85

Chart - Adapter (Class)

Variables	Constants
adaptee	EllipseFigure
adapter	EllipseFigureGeometricAdapter
client	HTMLTextAreaFigure
target	GeometricFigure
Operations	Signatures_28
Requests	Signatures_30
SpecificRequests	Signatures_31

Assignment - DetectionAssignment_84

Chart - Adapter (Class)

Variables	Constants
adaptee	DiamondFigure
adapter	DiamondFigureGeometricAdapter
client	HTMLTextAreaFigure
target	GeometricFigure
Operations	Signatures_28
Requests	Signatures_30
SpecificRequests	Signatures_29

Assignment - DetectionAssignment_666

Chart - Adapter (Class)

Variables	Constants
adaptee	CompositeFigure
adapter	StandardDrawing
client	ConnectionHandle
target	Drawing
Operations	Signatures_101
Requests	Signatures_52
SpecificRequests	Signatures_167

Assignment - DetectionAssignment_438

Chart - Adapter (Class)

Variables	Constants
adaptee	AbstractFigure
adapter	DecoratorFigure
client	GraphicalCompositeFigure
target	FigureChangeListener
Operations	Signatures_35
Requests	Signatures_37
SpecificRequests	Signatures_149

Assignment - DetectionAssignment_400

Chart - Adapter (Class)

Variables	Constants
adaptee	AbstractFigure
adapter	CompositeFigure
client	GraphicalCompositeFigure
target	FigureChangeListener
Operations	Signatures_35
Requests	Signatures_37
SpecificRequests	Signatures_149

Assignment - DetectionAssignment_368

Chart - Adapter (Class)

Variables	Constants
adaptee	DecoratorFigure
adapter	AnimationDecorator
client	Animator
target	Animatable
Operations	Signatures_136
Requests	Signatures_138
SpecificRequests	Signatures_137

Assignment - DetectionAssignment_115

Chart - Adapter (Class)

Variables	Constants
adaptee	TriangleFigure
adapter	TriangleFigureGeometricAdapter
client	HTMLTextAreaFigure
target	GeometricFigure
Operations	Signatures_28
Requests	Signatures_30
SpecificRequests	Signatures_29

Assignment - DetectionAssignment_113

Chart - Adapter (Class)

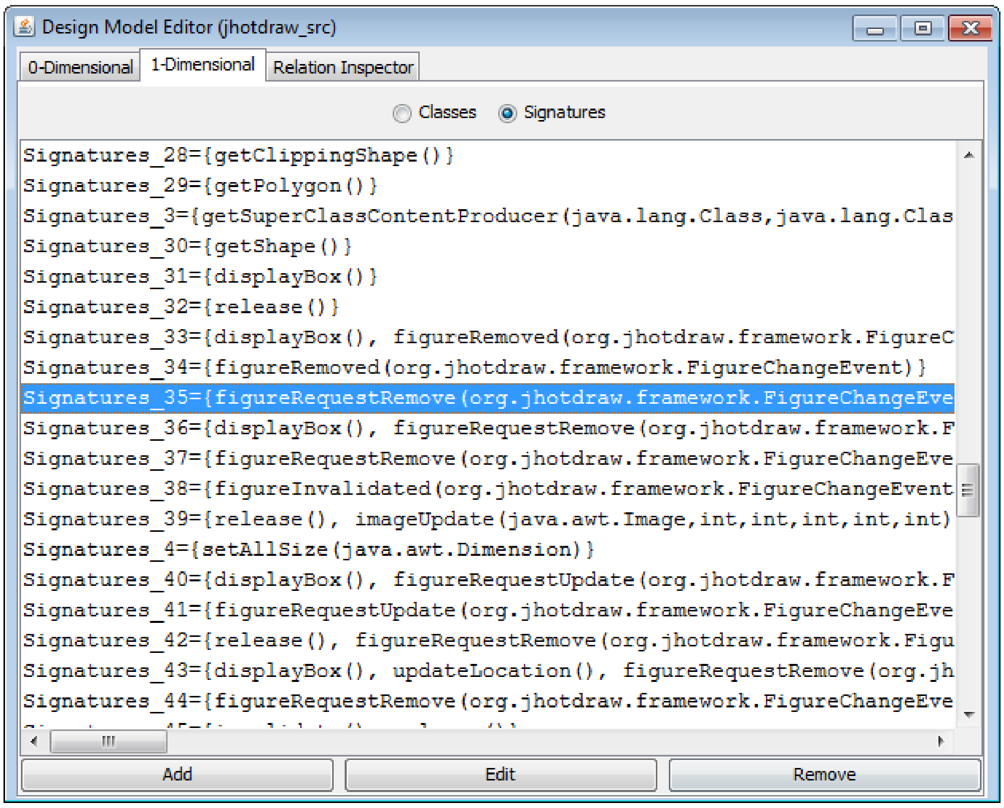
Variables	Constants
adaptee	PolygonFigure
adapter	PolygonFigureGeometricAdapter
client	HTMLTextAreaFigure
target	GeometricFigure
Operations	Signatures_28
Requests	Signatures_30
SpecificRequests	Signatures_65

Assignment - DetectionAssignment_114

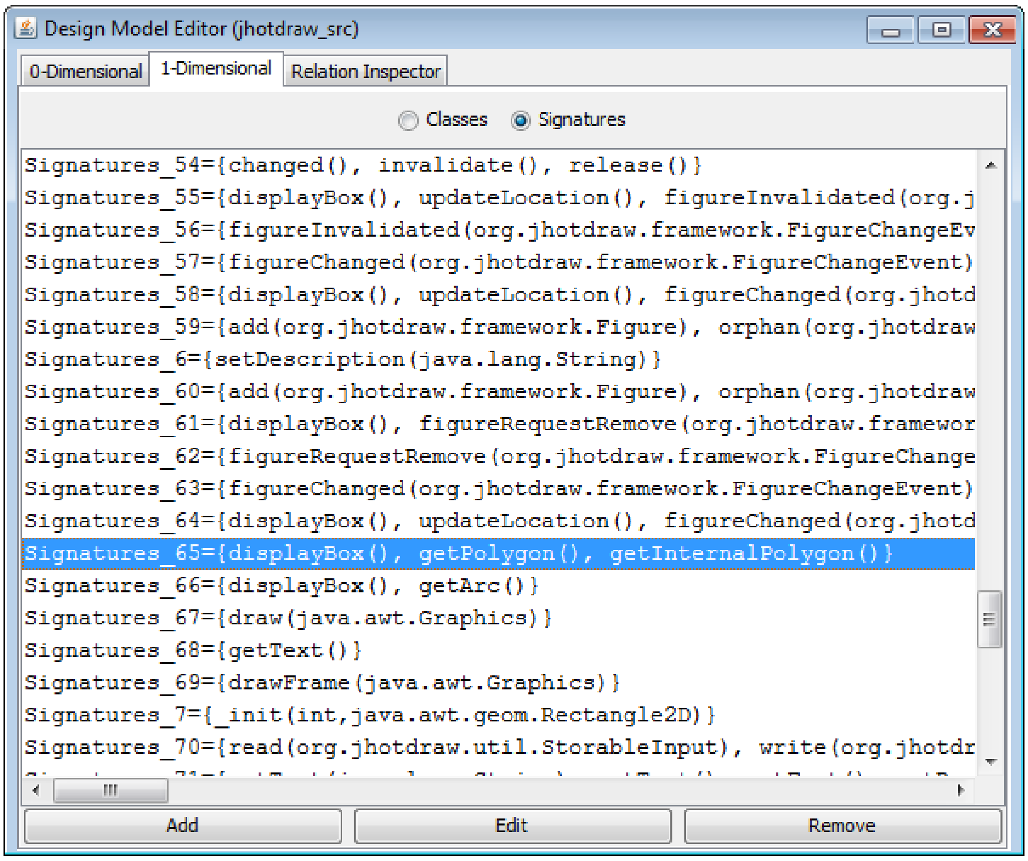
Chart - Adapter (Class)

Variables	Constants
adaptee	RoundRectangleFigure
adapter	RoundRectangleGeometricAdapter
client	HTMLTextAreaFigure
target	GeometricFigure
Operations	Signatures_28
Requests	Signatures_30
SpecificRequests	Signatures_66

Figure 136: assignments of Class Adapter detected in JHotdraw



```
Signatures_28={getClippingShape()}
Signatures_29={getPolygon()}
Signatures_3={getSuperClassContentProducer(java.lang.Class,java.lang.Clas
Signatures_30={getShape()}
Signatures_31={displayBox()}
Signatures_32={release()}
Signatures_33={displayBox(), figureRemoved(org.jhotdraw.framework.FigureC
Signatures_34={figureRemoved(org.jhotdraw.framework.FigureChangeEvent)}
Signatures_35={figureRequestRemove(org.jhotdraw.framework.FigureChangeEve
Signatures_36={displayBox(), figureRequestRemove(org.jhotdraw.framework.F
Signatures_37={figureRequestRemove(org.jhotdraw.framework.FigureChangeEve
Signatures_38={figureInvalidated(org.jhotdraw.framework.FigureChangeEvent
Signatures_39={release(), imageUpdate(java.awt.Image,int,int,int,int,int)
Signatures_4={setAllSize(java.awt.Dimension)}
Signatures_40={displayBox(), figureRequestUpdate(org.jhotdraw.framework.F
Signatures_41={figureRequestUpdate(org.jhotdraw.framework.FigureChangeEve
Signatures_42={release(), figureRequestRemove(org.jhotdraw.framework.Figu
Signatures_43={displayBox(), updateLocation(), figureRequestRemove(org.jh
Signatures_44={figureRequestRemove(org.jhotdraw.framework.FigureChangeEve
```



```
Signatures_54={changed(), invalidate(), release()}
Signatures_55={displayBox(), updateLocation(), figureInvalidated(org.j
Signatures_56={figureInvalidated(org.jhotdraw.framework.FigureChangeEv
Signatures_57={figureChanged(org.jhotdraw.framework.FigureChangeEvent)
Signatures_58={displayBox(), updateLocation(), figureChanged(org.jhotd
Signatures_59={add(org.jhotdraw.framework.Figure), orphan(org.jhotdraw
Signatures_6={setDescription(java.lang.String)}
Signatures_60={add(org.jhotdraw.framework.Figure), orphan(org.jhotdraw
Signatures_61={displayBox(), figureRequestRemove(org.jhotdraw.framewor
Signatures_62={figureRequestRemove(org.jhotdraw.framework.FigureChange
Signatures_63={figureChanged(org.jhotdraw.framework.FigureChangeEvent)
Signatures_64={displayBox(), updateLocation(), figureChanged(org.jhotd
Signatures_65={displayBox(), getPolygon(), getInternalPolygon()}
Signatures_66={displayBox(), getArc()}
Signatures_67={draw(java.awt.Graphics)}
Signatures_68={getText()}
Signatures_69={drawFrame(java.awt.Graphics)}
Signatures_7={_init(int,java.awt.geom.Rectangle2D)}
Signatures_70={read(org.jhotdraw.util.StorableInput), write(org.jhotdr
```

Figure 137: Auto-defined higher-dimension entities (Methods) (Adapter - JHotdraw)

Signatures Sets details

Signatures_29={getPolygon()}

Signatures_28={getClippingShape()}

Signatures_30={getShape()}

Signatures_31={displayBox()}

Signatures_35={figureRequestRemove(org.jhotdraw.framework.FigureChangeEvent), change()}

Signatures_37={figureRequestRemove(org.jhotdraw.framework.FigureChangeEvent), figureRequestUpdate(org.jhotdraw.framework.FigureChangeEvent)}

Signatures_52={add(org.jhotdraw.framework.Figure)}

Signatures_65={displayBox(), getPolygon(), getInternalPolygon()}

Signatures_66={displayBox(), getArc()}

Signatures_101={invokeStart(int,int,org.jhotdraw.framework.DrawingView)}

Signatures_136={run()}

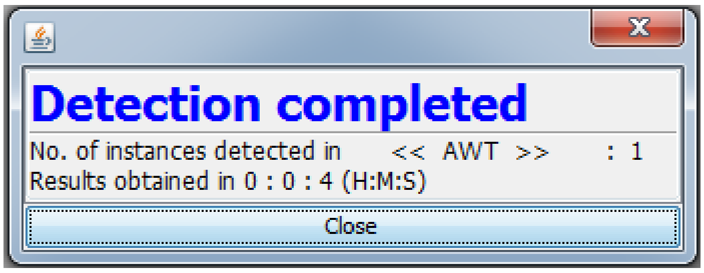
Signatures_137={displayBox(), moveBy(int,int), figures()}

Signatures_138={animationStep()}

Signatures_149={includes(org.jhotdraw.framework.Figure), displayBox(), invalidateRectangle(java.awt.Rectangle), listener()}

Signatures_167={containsFigure(org.jhotdraw.framework.Figure), add(org.jhotdraw.framework.Figure), _addToQuadTree(org.jhotdraw.framework.Figure), figureRequestUpdate(org.jhotdraw.framework.FigureChangeEvent)}

Class Adapter pattern in Java.AWT



Detection completed

No. of instances detected in << AWT >> : 1
Results obtained in 0 : 0 : 4 (H:M:S)

Figure 138: Results of detecting Class Adapter pattern in Java.AWT

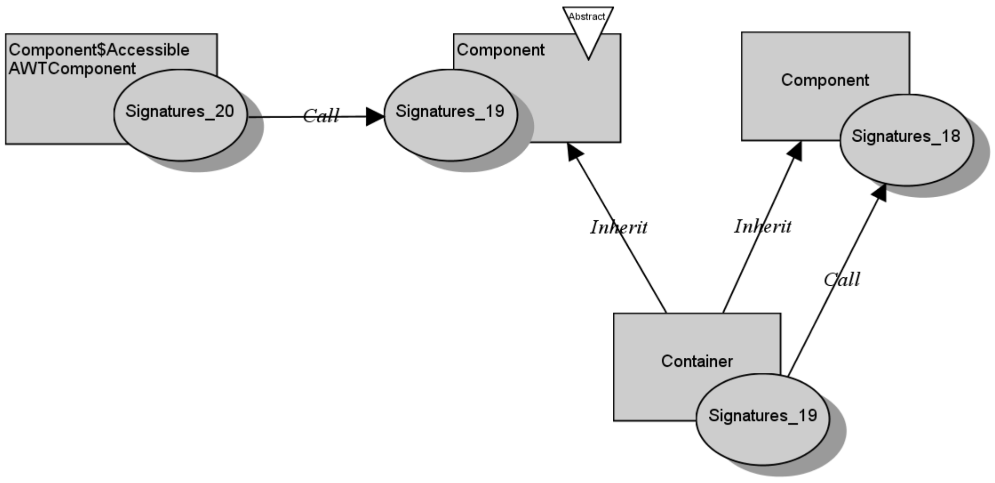
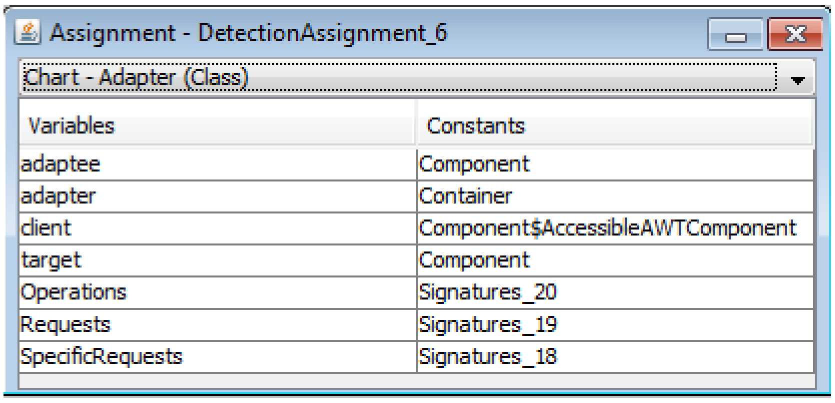
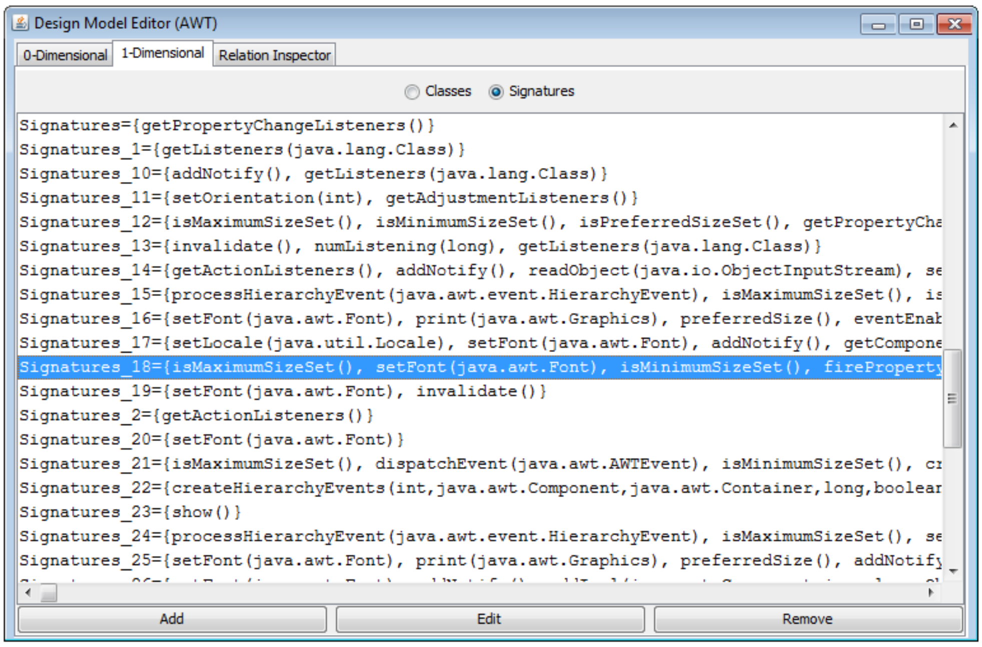


Figure 139: Codechart: An instance of Class Adapter pattern detected in Java.AWT



Variables	Constants
adaptee	Component
adapter	Container
client	Component$AccessibleAWTComponent
target	Component
Operations	Signatures_20
Requests	Signatures_19
SpecificRequests	Signatures_18

Figure 140: An assignment of Class Adapter pattern detected in Java.AWT



( ) Classes [✓ Signatures]

```
Signatures={getPropertyChangeListeners()}
Signatures_1={getListeners(java.lang.Class)}
Signatures_10={addNotify(), getListeners(java.lang.Class)}
Signatures_11={setOrientation(int), getAdjustmentListeners()}
Signatures_12={isMaximumSizeSet(), isMinimumSizeSet(), isPreferredSizeSet(), getPropertyCha
Signatures_13={invalidate(), numListening(long), getListeners(java.lang.Class)}
Signatures_14={getActionListeners(), addNotify(), readObject(java.io.ObjectInputStream), se
Signatures_15={processHierarchyEvent(java.awt.event.HierarchyEvent), isMaximumSizeSet(), is
Signatures_16={setFont(java.awt.Font), print(java.awt.Graphics), preferredSize(), eventEnak
Signatures_17={setLocale(java.util.Locale), setFont(java.awt.Font), addNotify(), getCompone
Signatures_18={isMaximumSizeSet(), setFont(java.awt.Font), isMinimumSizeSet(), fireProperty
Signatures_19={setFont(java.awt.Font), invalidate()}
Signatures_2={getActionListeners()}
Signatures_20={setFont(java.awt.Font)}
Signatures_21={isMaximumSizeSet(), dispatchEvent(java.awt.AWTEvent), isMinimumSizeSet(), cr
Signatures_22={createHierarchyEvents(int,java.awt.Component,java.awt.Container,long,boolear
Signatures_23={show()}
Signatures_24={processHierarchyEvent(java.awt.event.HierarchyEvent), isMaximumSizeSet(), se
Signatures_25={setFont(java.awt.Font), print(java.awt.Graphics), preferredSize(), addNotify
```

Figure 141: Auto-defined higher-dimension entities (Methods) (Class Adapter pattern - Java.AWT)

Signatures Set details

Signatures_18={isMaximumSizeSet(), setFont(java.awt.Font), isMinimumSizeSet(), firePropertyChange(java.lang.String,java.lang.Object,java.lang.Object), isPreferredSizeSet(), getFont(), invalidate(), getTreeLock()}
 	
Signatures_19={setFont(java.awt.Font), invalidate()}
 	
Signatures_20={setFont(java.awt.Font)}

Strategy pattern Case Studies

It is important to report that case studies of the strategy pattern (shown in Eden [38]) could not be detected as seen in Figure 143 and Figure 149. This is due to the client participant shown in the pattern Codechart. However, as this participant should be a user defined class, the participant in the Codechart could be removed. Figure 142 shows the pattern Codechart without the client participant. Having this new Codechart, the pattern could be detected and the Figure 144 and Figure 150 shows the detection results of strategy pattern in JRefactory and JHotdraw respectively.

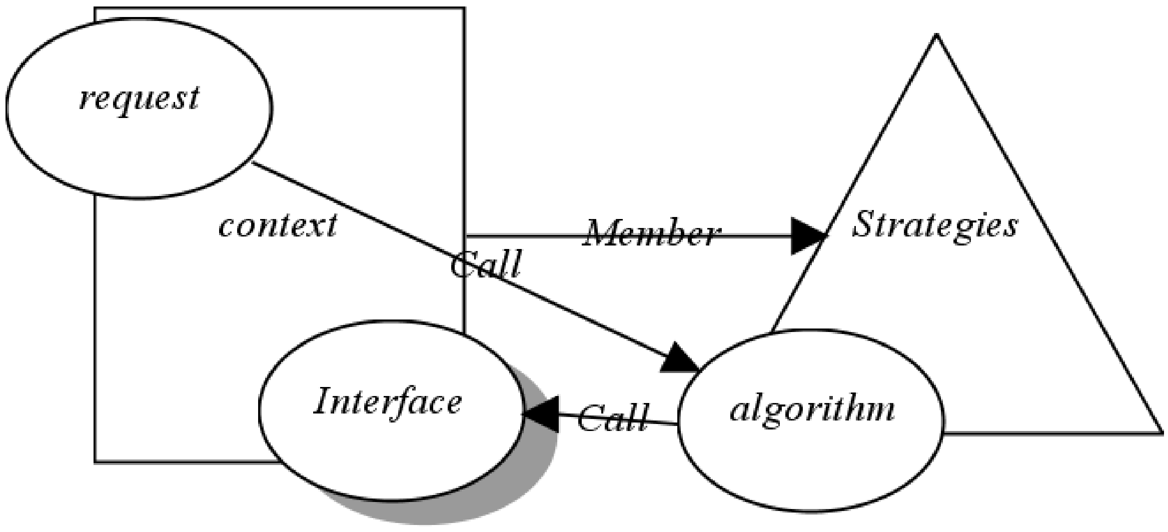


Figure 142: Codechart: Strategy pattern (without Client class variable)

Strategy pattern in JRefactory

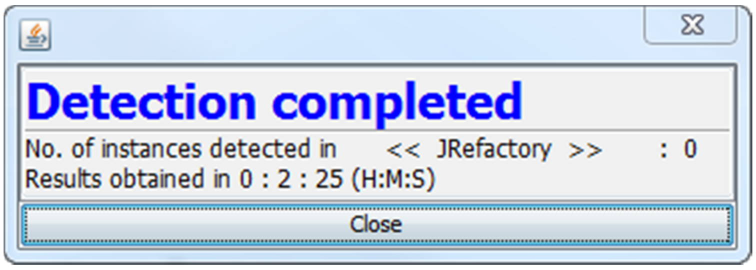

Detection completed
No. of instances detected in << JRefactory >> : 0
Results obtained in 0 : 2 : 25 (H:M:S)

Figure 143: Results of detecting Strategy in JRefactory

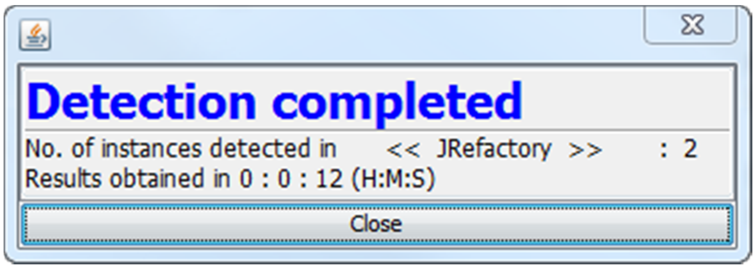

Detection completed
No. of instances detected in << JRefactory >> : 2
Results obtained in 0 : 0 : 12 (H:M:S)

Figure 144: Results of detecting Strategy (without Client class variable) in JRefactory

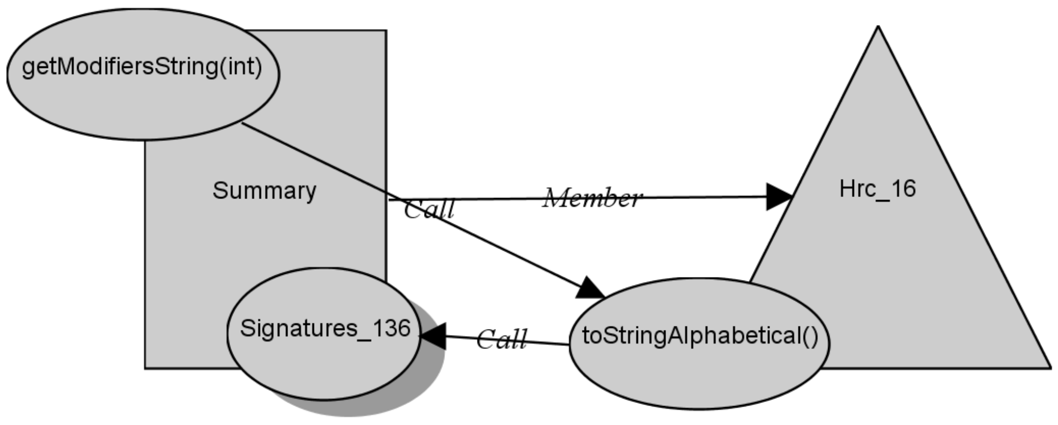


Figure 145: Codechart: An instance of Strategy pattern (without Client class variable) detected in JRefactory

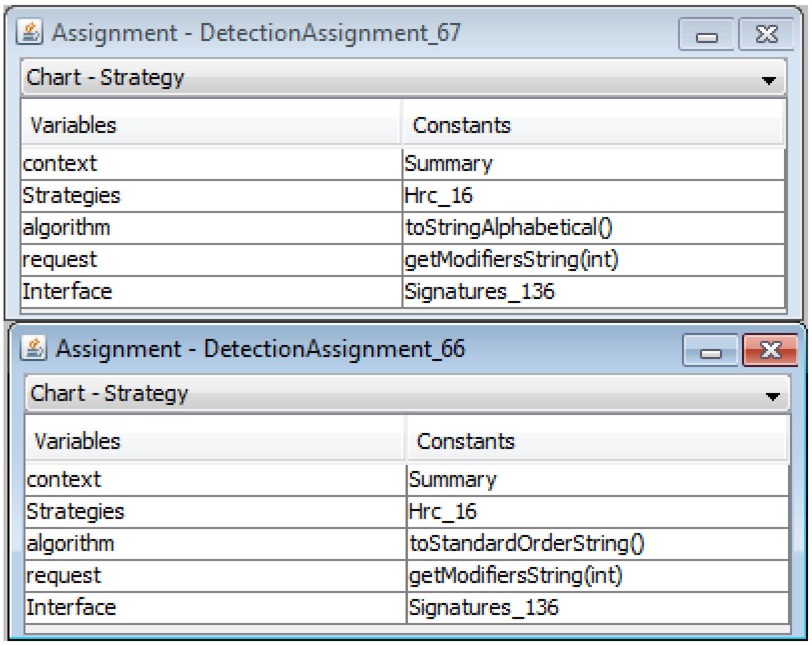

Assignment - DetectionAssignment_67

Chart - Strategy

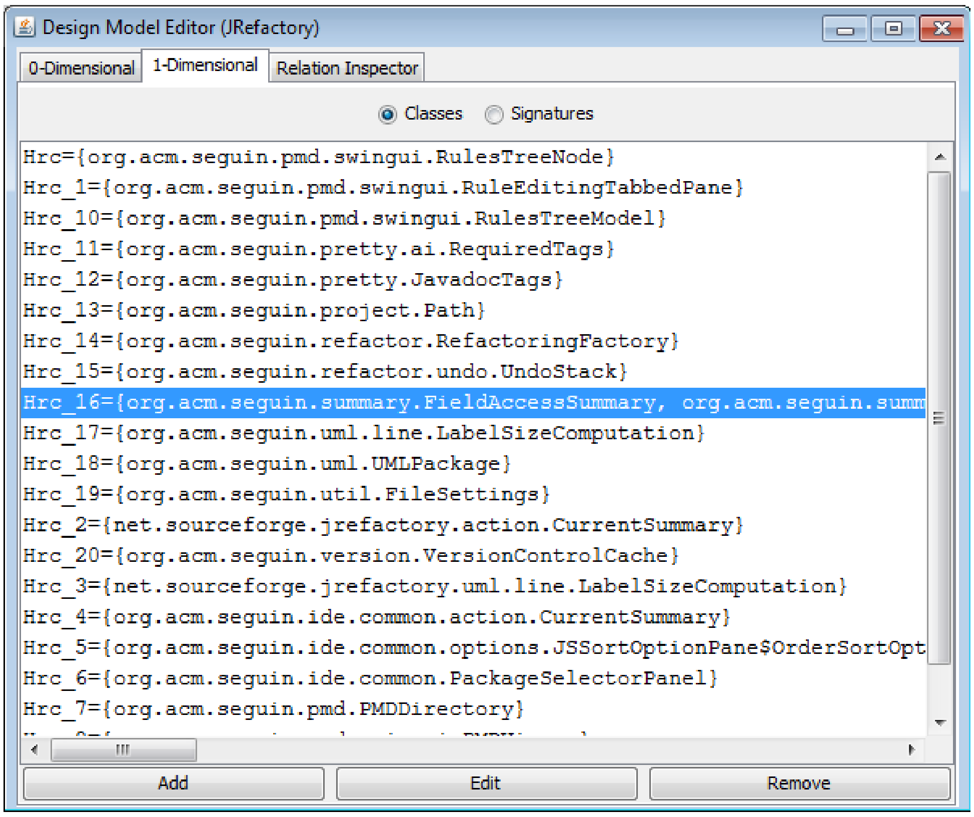
Variables	Constants
context	Summary
Strategies	Hrc_16
algorithm	toStringAlphabetical()
request	getModifiersString(int)
Interface	Signatures_136

Assignment - DetectionAssignment_66

Chart - Strategy

Variables	Constants
context	Summary
Strategies	Hrc_16
algorithm	toStandardOrderString()
request	getModifiersString(int)
Interface	Signatures_136

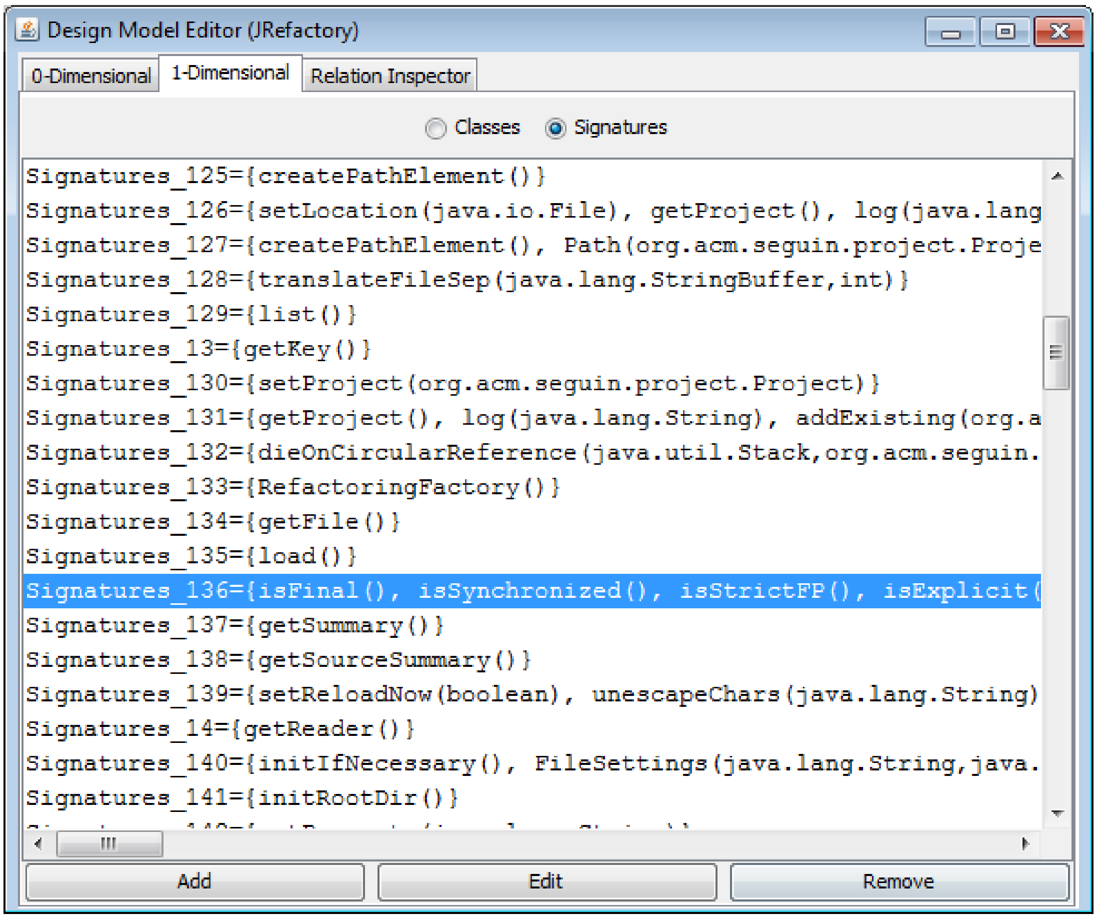
Figure 146: assignments of Strategy (without Client class variable) detected in JRefactory



[✓ Classes] Signatures

```
Hrc={org.acm.seguin.pmd.swingui.RulesTreeNode}
Hrc_1={org.acm.seguin.pmd.swingui.RuleEditingTabbedPane}
Hrc_10={org.acm.seguin.pmd.swingui.RulesTreeModel}
Hrc_11={org.acm.seguin.pretty.ai.RequiredTags}
Hrc_12={org.acm.seguin.pretty.JavadocTags}
Hrc_13={org.acm.seguin.project.Path}
Hrc_14={org.acm.seguin.refactor.RefactoringFactory}
Hrc_15={org.acm.seguin.refactor.undo.UndoStack}
Hrc_16={org.acm.seguin.summary.FieldAccessSummary, org.acm.seguin.summ
Hrc_17={org.acm.seguin.uml.line.LabelSizeComputation}
Hrc_18={org.acm.seguin.uml.UMLPackage}
Hrc_19={org.acm.seguin.util.FileSettings}
Hrc_2={net.sourceforge.jrefactory.action.CurrentSummary}
Hrc_20={org.acm.seguin.version.VersionControlCache}
Hrc_3={net.sourceforge.jrefactory.uml.line.LabelSizeComputation}
Hrc_4={org.acm.seguin.ide.common.action.CurrentSummary}
Hrc_5={org.acm.seguin.ide.common.options.JSSortOptionPane$OrderSortOpt
Hrc_6={org.acm.seguin.ide.common.PackageSelectorPanel}
Hrc_7={org.acm.seguin.pmd.PMDDirectory}
```

Figure 147: Auto-defined higher-dimension entities (Hierarchies) (Strategy -without Client class variable - JRefactory)

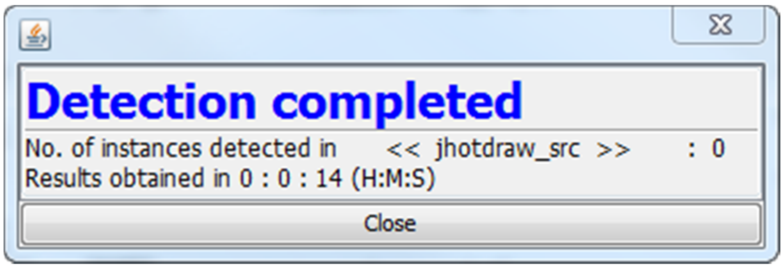


```
Signatures_125={createPathElement()}
Signatures_126={setLocation(java.io.File), getProject(), log(java.lang
Signatures_127={createPathElement(), Path(org.acm.seguin.project.Proje
Signatures_128={translateFileSep(java.lang.StringBuffer,int)}
Signatures_129={list()}
Signatures_13={getKey()}
Signatures_130={setProject(org.acm.seguin.project.Project)}
Signatures_131={getProject(), log(java.lang.String), addExisting(org.a
Signatures_132={dieOnCircularReference(java.util.Stack,org.acm.seguin.
Signatures_133={RefactoringFactory()}
Signatures_134={getFile()}
Signatures_135={load()}
Signatures_136={isFinal(), isSynchronized(), isStrictFP(), isExplicit(
Signatures_137={getSummary()}
Signatures_138={getSourceSummary()}
Signatures_139={setReloadNow(boolean), unescapeChars(java.lang.String)
Signatures_14={getReader()}
Signatures_140={initIfNecessary(), FileSettings(java.lang.String,java.
Signatures_141={initRootDir()}
```

Figure 148: Auto-defined higher-dimension entities (Methods) (Strategy -without Client class variable - JRefactory)

Signatures Set details
Signatures_136={isFinal(), isSynchronized(), isStrictFP(), isExplicit(), isNative(), isPublic(), isVolatile(), isAbstract(), isTransient(), isPrivate(), isInterface(), isProtected(), isStatic()}
Hierarchies details
Hrc_16={org.acm.seguin.summary.FieldAccessSummary, org.acm.seguin.summary.TypeSummary, org.acm.seguin.summary.FileSummary, org.acm.seguin.summary.MethodSummary, org.acm.seguin.summary.Summary, org.acm.seguin.summary.PackageSummary, org.acm.seguin.summary.ImportSummary, org.acm.seguin.summary.TypeDeclSummary, org.acm.seguin.summary.MessageSendSummary, org.acm.seguin.summary.VariableSummary}

Strategy pattern in JHotdraw

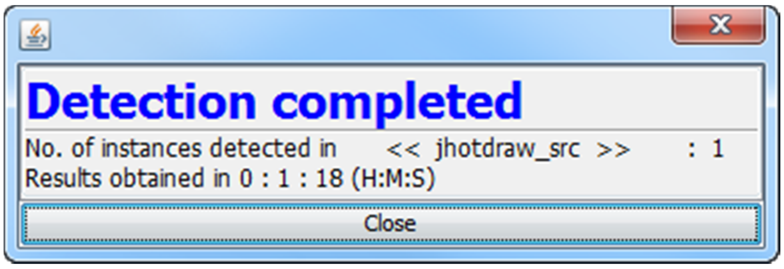


Detection completed

No. of instances detected in << jhotdraw_src >> : 0

Results obtained in 0 : 0 : 14 (H:M:S)

Figure 149: Results of detecting Strategy in JHotdraw



Detection completed

No. of instances detected in << jhotdraw_src >> : 1

Results obtained in 0 : 1 : 18 (H:M:S)

Figure 150: Results of detecting Strategy (without Client class variable) in JHotdraw

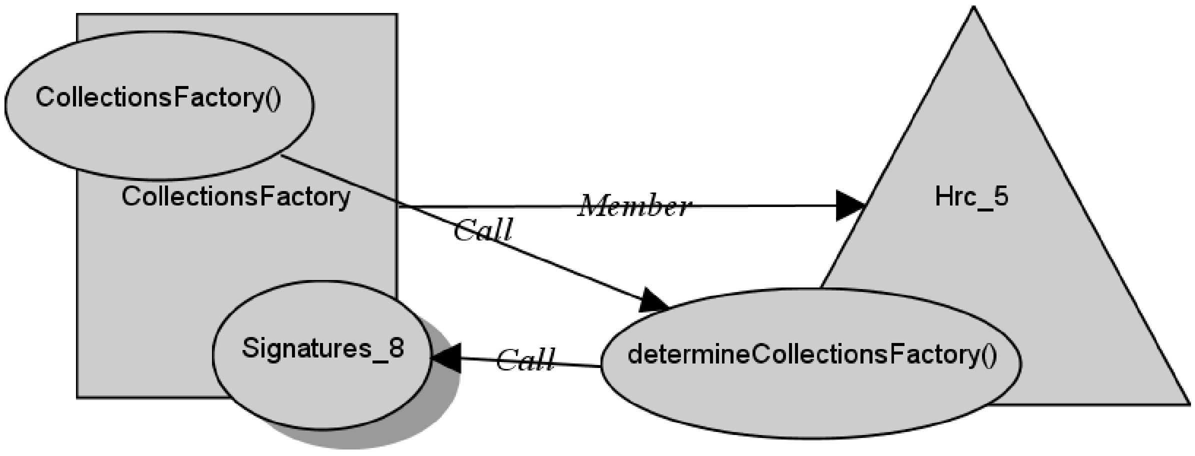
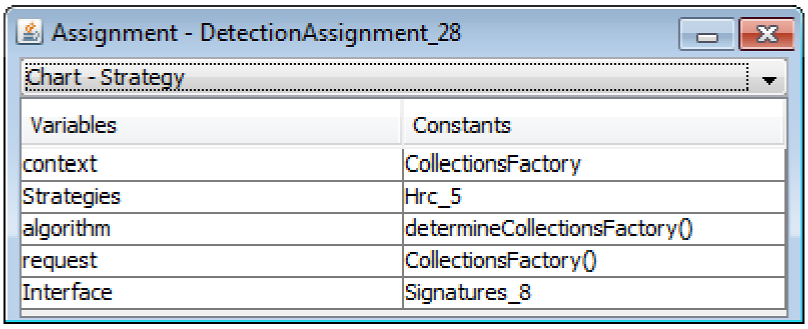


Figure 151: Codechart: An instance of Strategy pattern (without Client class variable) detected in JHotdraw

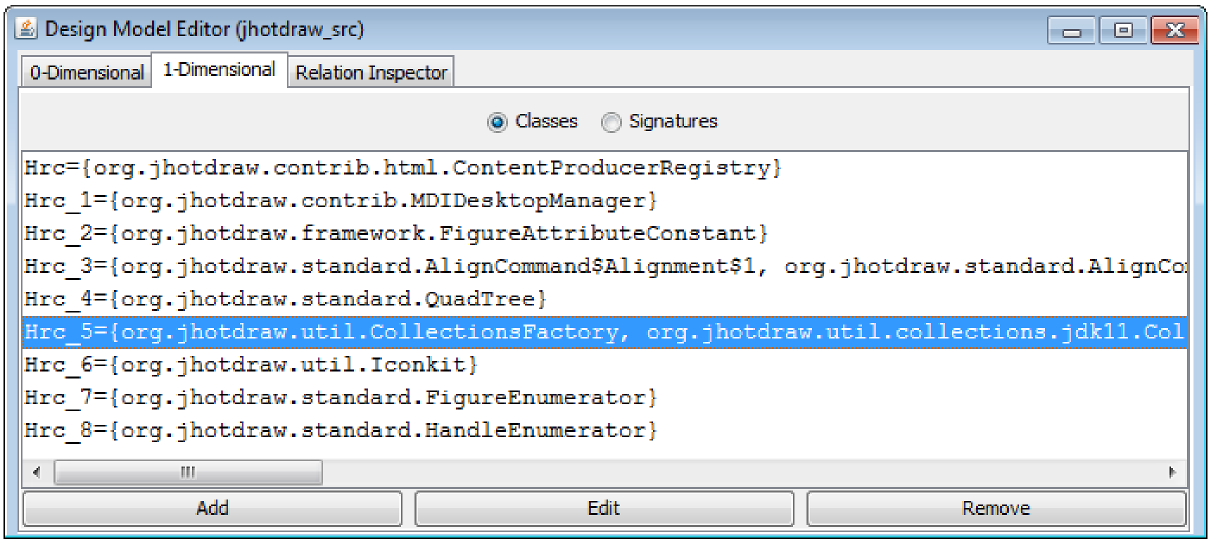


Assignment - DetectionAssignment_28

Chart - Strategy

Variables	Constants
context	CollectionsFactory
Strategies	Hrc_5
algorithm	determineCollectionsFactory()
request	CollectionsFactory()
Interface	Signatures_8

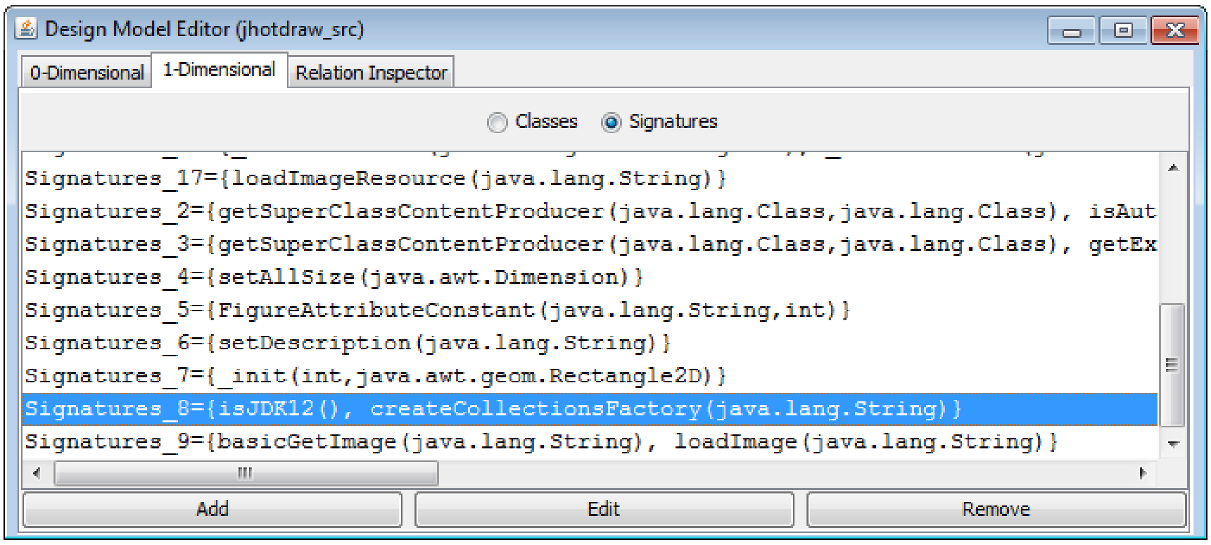
Figure 152: an assignment of Strategy (without Client class variable) detected in JHotdraw



Classes [✓] Signatures

```
Hrc={org.jhotdraw.contrib.html.ContentProducerRegistry}
Hrc_1={org.jhotdraw.contrib.MDIDesktopManager}
Hrc_2={org.jhotdraw.framework.FigureAttributeConstant}
Hrc_3={org.jhotdraw.standard.AlignCommand$Alignment$1, org.jhotdraw.standard.AlignCo
Hrc_4={org.jhotdraw.standard.QuadTree}
Hrc_5={org.jhotdraw.util.CollectionsFactory, org.jhotdraw.util.collections.jdk11.Col
Hrc_6={org.jhotdraw.util.Iconkit}
Hrc_7={org.jhotdraw.standard.FigureEnumerator}
Hrc_8={org.jhotdraw.standard.HandleEnumerator}
```

Figure 153: Auto-defined higher-dimension entities (Hierarchies) (Strategy -without Client class variable - JHotdraw)



Classes [✓ Signatures]

```
Signatures_17={loadImageResource(java.lang.String)}
Signatures_2={getSuperClassContentProducer(java.lang.Class,java.lang.Class), isAut
Signatures_3={getSuperClassContentProducer(java.lang.Class,java.lang.Class), getEx
Signatures_4={setAllSize(java.awt.Dimension)}
Signatures_5={FigureAttributeConstant(java.lang.String,int)}
Signatures_6={setDescription(java.lang.String)}
Signatures_7={_init(int,java.awt.geom.Rectangle2D)}
Signatures_8={isJDK12(), createCollectionsFactory(java.lang.String)}
Signatures_9={basicGetImage(java.lang.String), loadImage(java.lang.String)}
```

Figure 154: Auto-defined higher-dimension entities (Methods) (Strategy -without Client class variable - JHotdraw)

Signatures Set details
Signatures_8={isJDK12(), createCollectionsFactory(java.lang.String)}
Hierarchies details

Hrc_5={org.jhotdraw.util.CollectionsFactory,
org.jhotdraw.util.collections.jdk11.CollectionsFactoryJDK11,
org.jhotdraw.util.collections.jdk12.CollectionsFactoryJDK12}

Strategy pattern in Java.AWT

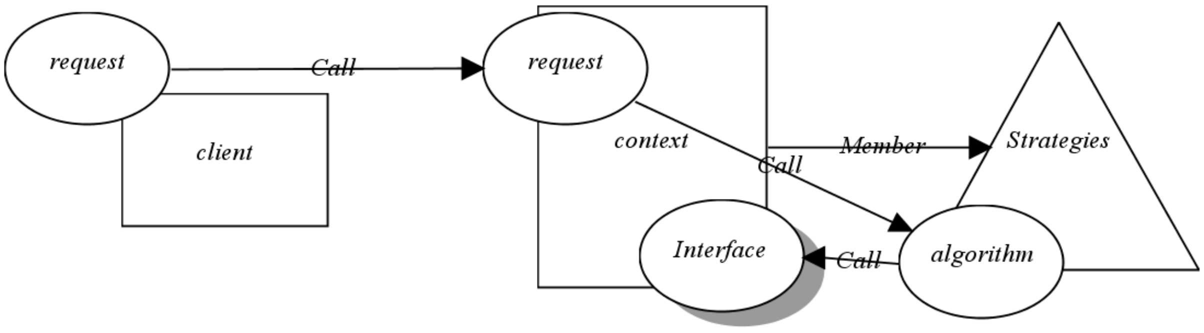


Figure 155: Codechart: Strategy pattern [38]

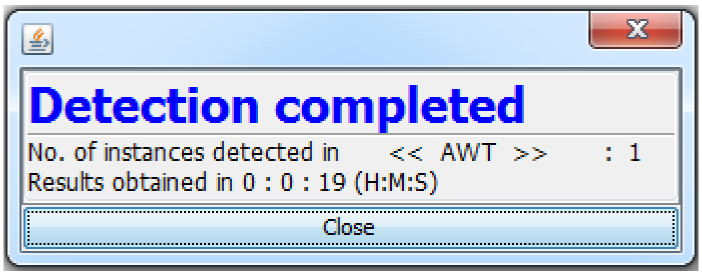

Detection completed
No. of instances detected in << AWT >> : 1
Results obtained in 0 : 0 : 19 (H:M:S)

Figure 156: Results of detecting Strategy in Java.AWT

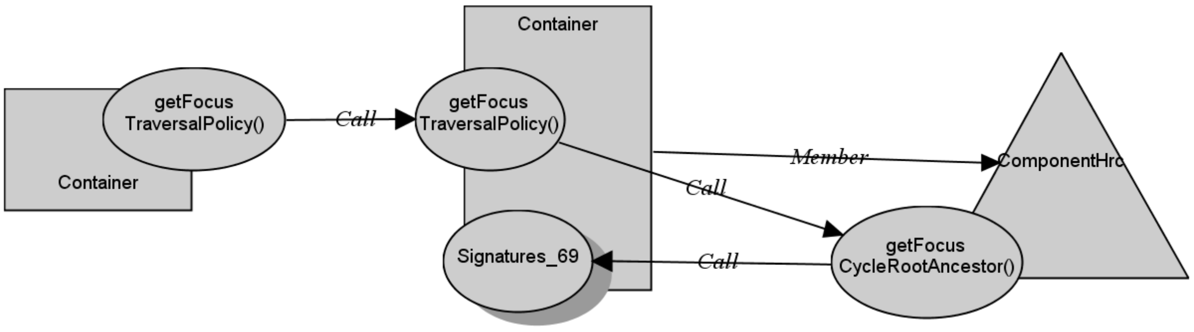
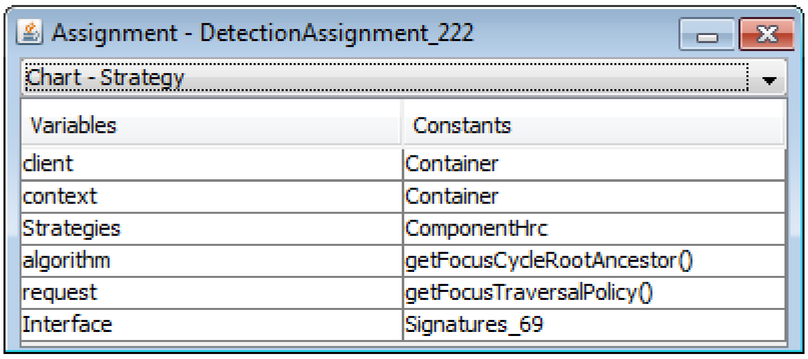
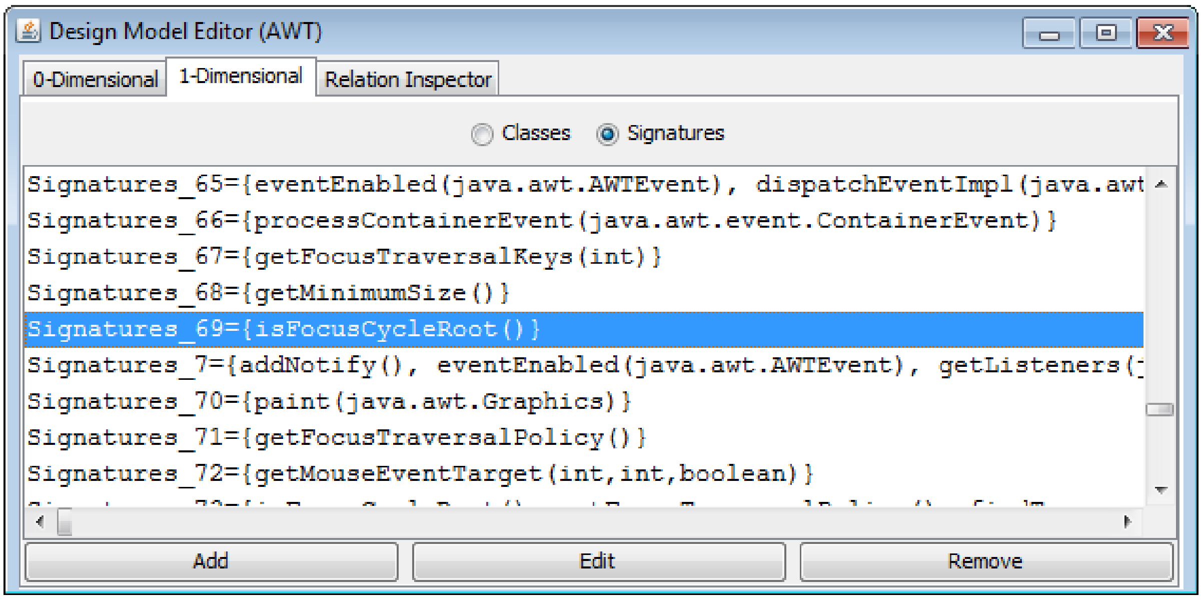

Container
getFocus TraversalPolicy()
Call
getFocus TraversalPolicy()
Container
Member
ComponentHrc
Call
Signatures_69
Call
getFocus CycleRootAncestor()

Figure 157: Codechart: An instance of Strategy pattern detected in Java.AWT



Variables	Constants
client	Container
context	Container
Strategies	ComponentHrc
algorithm	getFocusCycleRootAncestor()
request	getFocusTraversalPolicy()
Interface	Signatures_69

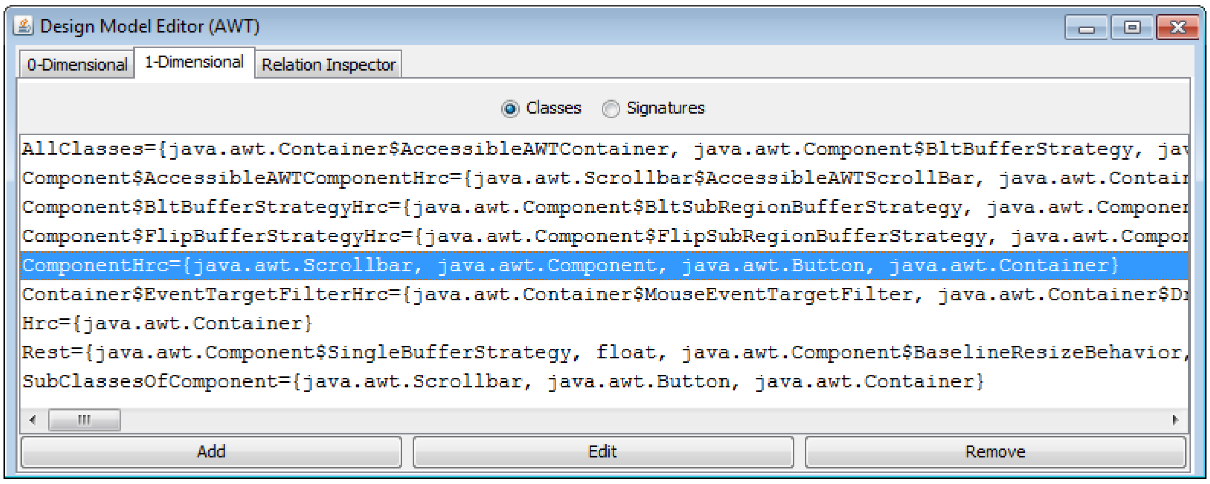
Figure 158: An assignment of Strategy detected in Java.AWT



Classes [✓ Signatures]

```
Signatures_65={eventEnabled(java.awt.AWTEvent), dispatchEventImpl(java.awt
Signatures_66={processContainerEvent(java.awt.event.ContainerEvent)}
Signatures_67={getFocusTraversalKeys(int)}
Signatures_68={getMinimumSize()}
Signatures_69={isFocusCycleRoot()}
Signatures_7={addNotify(), eventEnabled(java.awt.AWTEvent), getListeners(
Signatures_70={paint(java.awt.Graphics)}
Signatures_71={getFocusTraversalPolicy()}
Signatures_72={getMouseEventTarget(int,int,boolean)}
```

Figure 159: Auto-defined higher-dimension entities (Hierarchies) (Strategy - Java.AWT)



[✓ Classes] Signatures

```
AllClasses={java.awt.Container$AccessibleAWTContainer, java.awt.Component$BltBufferStrategy, ja
Component$AccessibleAWTComponentHrc={java.awt.Scrollbar$AccessibleAWTScrollBar, java.awt.Contai
Component$BltBufferStrategyHrc={java.awt.Component$BltSubRegionBufferStrategy, java.awt.Compone
Component$FlipBufferStrategyHrc={java.awt.Component$FlipSubRegionBufferStrategy, java.awt.Compo
ComponentHrc={java.awt.Scrollbar, java.awt.Component, java.awt.Button, java.awt.Container}
Container$EventTargetFilterHrc={java.awt.Container$MouseEventTargetFilter, java.awt.Container$D
Hrc={java.awt.Container}
Rest={java.awt.Component$SingleBufferStrategy, float, java.awt.Component$BaselineResizeBehavior,
SubClassesOfComponent={java.awt.Scrollbar, java.awt.Button, java.awt.Container}
```

Figure 160: Auto-defined higher-dimension entities (Methods) (Strategy - Java.AWT)

Factory Method Case Studies

Factory Method pattern in JHotdraw

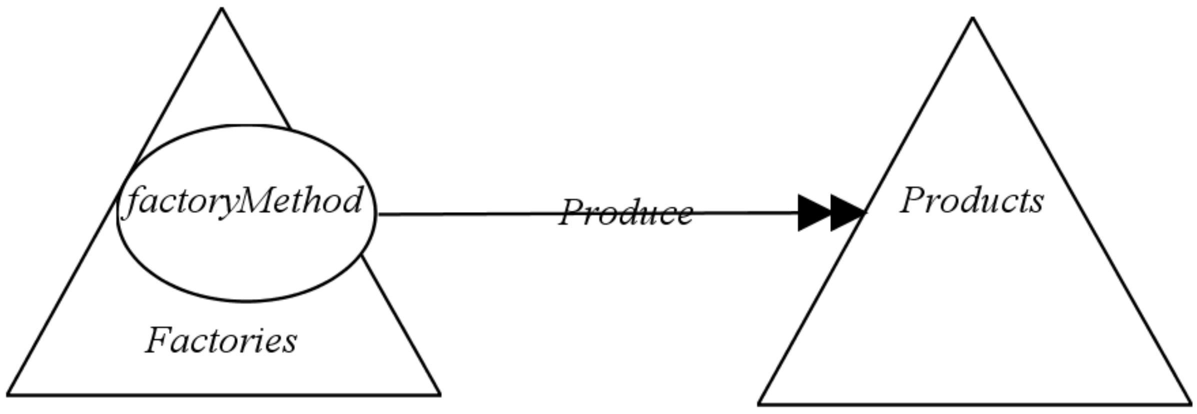
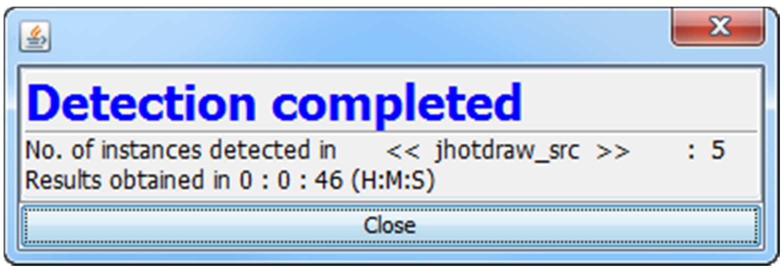


Figure 161: Codechart: Factory Method pattern [38]



Detection completed

No. of instances detected in << jhotdraw_src >> : 5
Results obtained in 0 : 0 : 46 (H:M:S)

Figure 162: Results of detecting Factory Method in JHotdraw

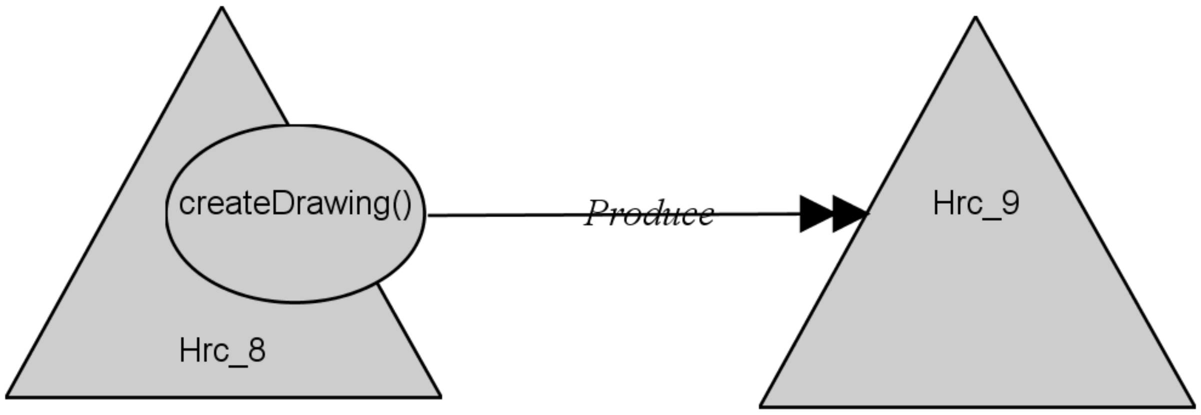
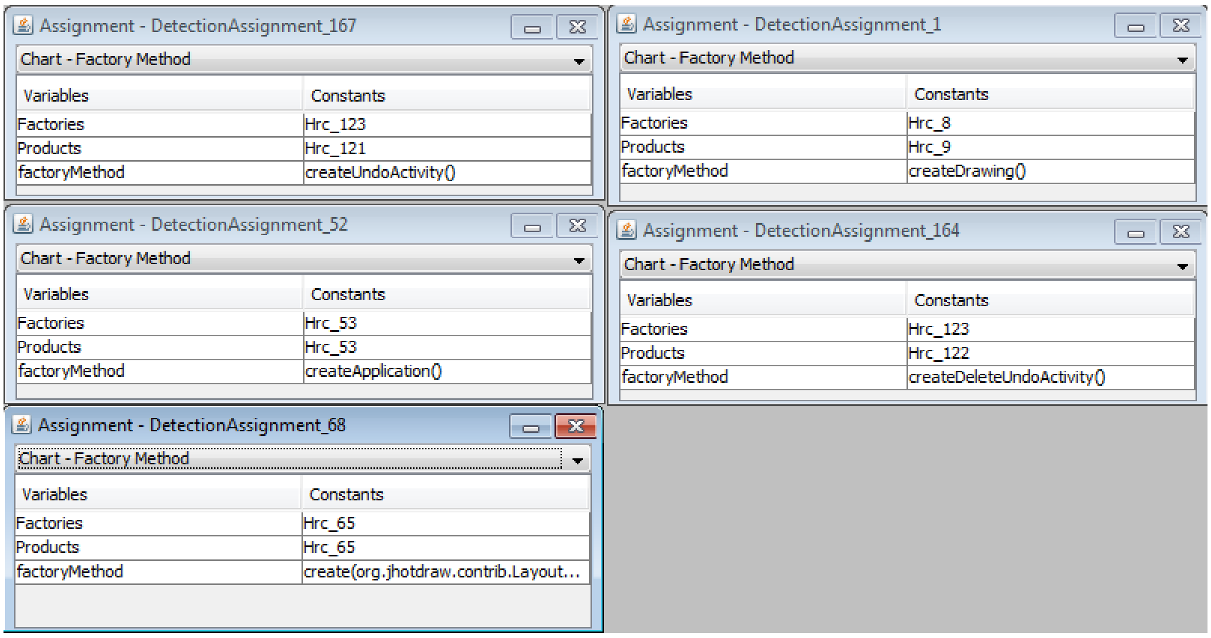


Figure 163: Codechart: An instance of Factory Method pattern detected in JHotdraw



Assignment - DetectionAssignment_167

Chart - Factory Method

Variables	Constants
Factories	Hrc_123
Products	Hrc_121
factoryMethod	createUndoActivity()

Assignment - DetectionAssignment_1

Chart - Factory Method

Variables	Constants
Factories	Hrc_8
Products	Hrc_9
factoryMethod	createDrawing()

Assignment - DetectionAssignment_52

Chart - Factory Method

Variables	Constants
Factories	Hrc_53
Products	Hrc_53
factoryMethod	createApplication()

Assignment - DetectionAssignment_164

Chart - Factory Method

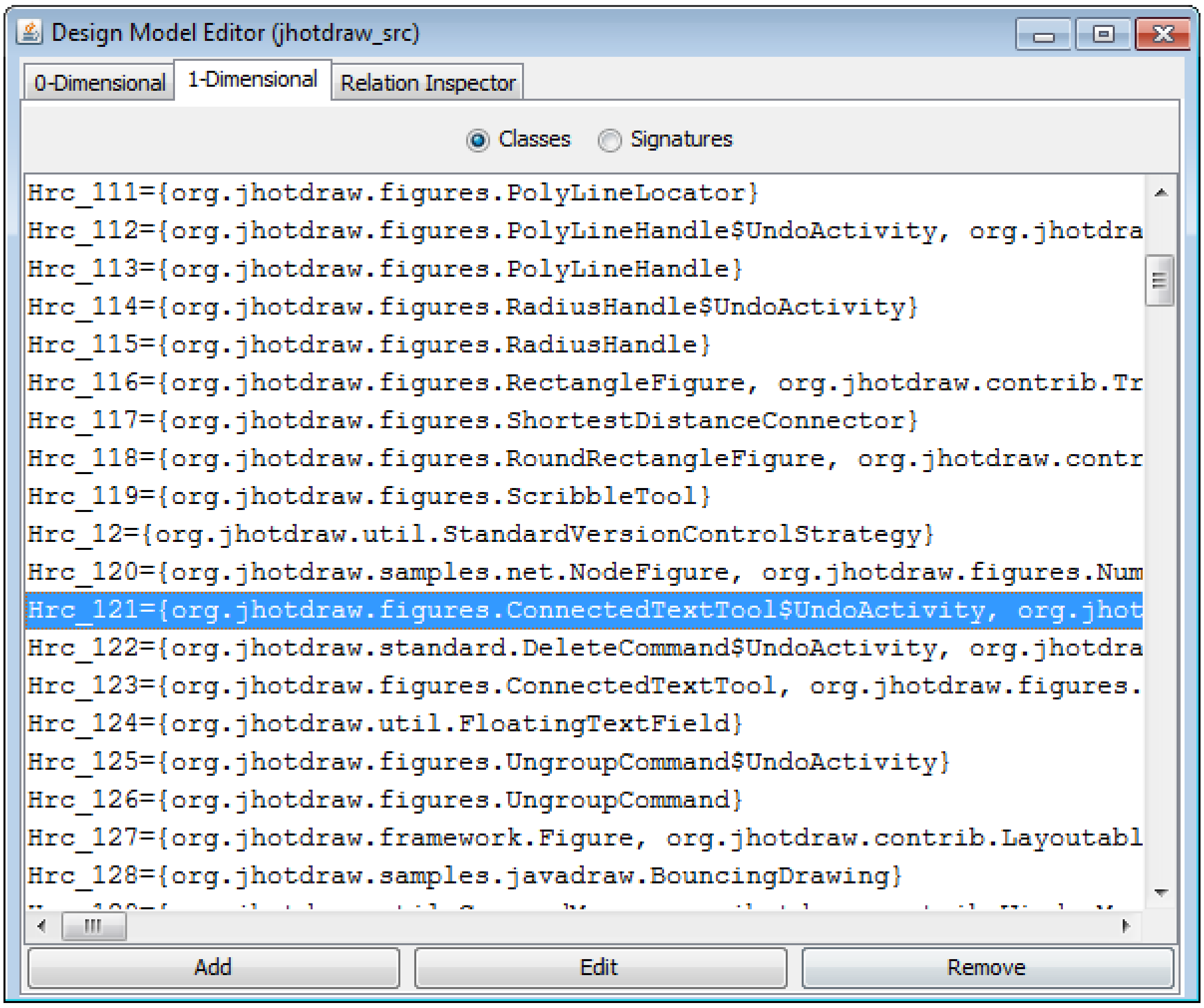
Variables	Constants
Factories	Hrc_123
Products	Hrc_122
factoryMethod	createDeleteUndoActivity()

Assignment - DetectionAssignment_68

Chart - Factory Method

Variables	Constants
Factories	Hrc_65
Products	Hrc_65
factoryMethod	create(org.jhotdraw.contrib.Layout...

Figure 164: Assignments of Factory Method detected in JHotdraw



[✓ Classes] Signatures

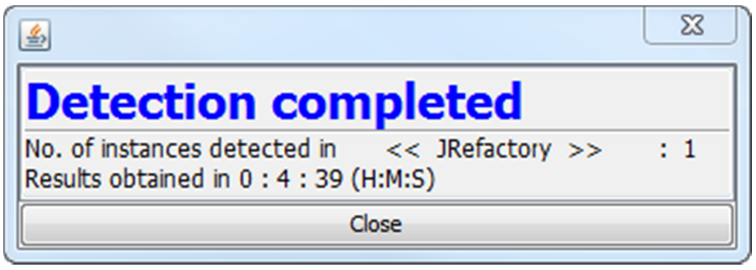
```
Hrc_111={org.jhotdraw.figures.PolyLineLocator}
Hrc_112={org.jhotdraw.figures.PolyLineHandle$UndoActivity, org.jhotdra
Hrc_113={org.jhotdraw.figures.PolyLineHandle}
Hrc_114={org.jhotdraw.figures.RadiusHandle$UndoActivity}
Hrc_115={org.jhotdraw.figures.RadiusHandle}
Hrc_116={org.jhotdraw.figures.RectangleFigure, org.jhotdraw.contrib.Tr
Hrc_117={org.jhotdraw.figures.ShortestDistanceConnector}
Hrc_118={org.jhotdraw.figures.RoundRectangleFigure, org.jhotdraw.contr
Hrc_119={org.jhotdraw.figures.ScribbleTool}
Hrc_12={org.jhotdraw.util.StandardVersionControlStrategy}
Hrc_120={org.jhotdraw.samples.net.NodeFigure, org.jhotdraw.figures.Num
Hrc_121={org.jhotdraw.figures.ConnectedTextTool$UndoActivity, org.jhot
Hrc_122={org.jhotdraw.standard.DeleteCommand$UndoActivity, org.jhotdra
Hrc_123={org.jhotdraw.figures.ConnectedTextTool, org.jhotdraw.figures.
Hrc_124={org.jhotdraw.util.FloatingTextField}
Hrc_125={org.jhotdraw.figures.UngroupCommand$UndoActivity}
Hrc_126={org.jhotdraw.figures.UngroupCommand}
Hrc_127={org.jhotdraw.framework.Figure, org.jhotdraw.contrib.Layoutabl
Hrc_128={org.jhotdraw.samples.javadraw.BouncingDrawing}
```

Figure 165: Auto-defined higher-dimension entities (Hierarchies) (Factory Method - JHotdraw)

Hierarchies details

Hrc_8={org.jhotdraw.samples.javadraw.JavaDrawApplet, org.jhotdraw.samples.nothing.NothingApplet, org.jhotdraw.samples.pert.PertApplet, org.jhotdraw.applet.DrawApplet}
 	
 Hrc_9={org.jhotdraw.samples.javadraw.BouncingDrawing, org.jhotdraw.standard.StandardDrawing}
 	
 Hrc_53={org.jhotdraw.samples.javadraw.JavaDrawApp, org.jhotdraw.contrib.MDI_DrawApplication}
 	
 Hrc_65={org.jhotdraw.contrib.SimpleLayouter, org.jhotdraw.contrib.StandardLayouter}
 	
 Hrc_121={org.jhotdraw.figures.ConnectedTextTool$UndoActivity, org.jhotdraw.figures.TextTool$UndoActivity}
 	
 Hrc_122={org.jhotdraw.standard.DeleteCommand$UndoActivity, org.jhotdraw.figures.ConnectedTextTool$DeleteUndoActivity}
 	
 Hrc_123={org.jhotdraw.figures.ConnectedTextTool, org.jhotdraw.figures.TextTool}

Factory Method pattern in JRefactory


Detection completed

```
No. of instances detected in     << JRefactory >>     : 1
Results obtained in 0 : 4 : 39 (H:M:S)
```

Figure 166: Results of detecting Factory Method in JRefactory

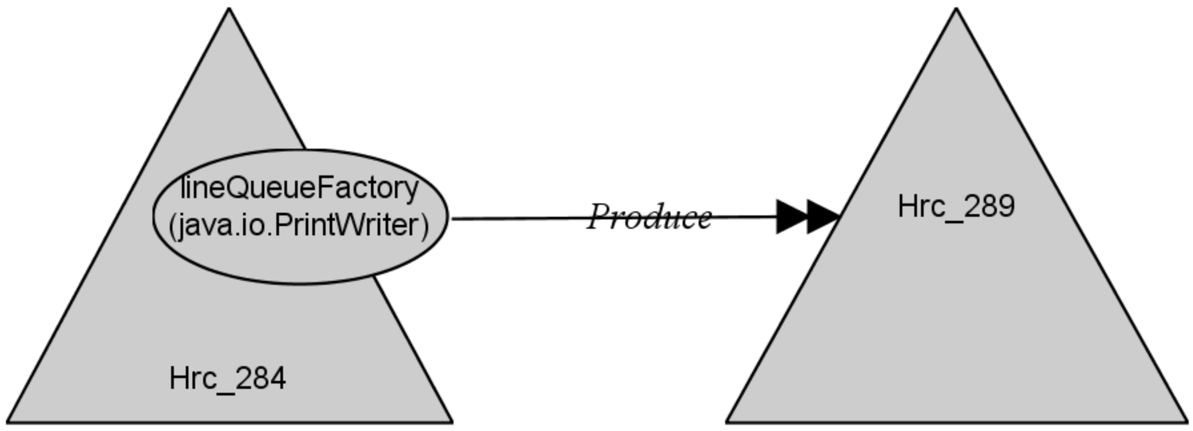
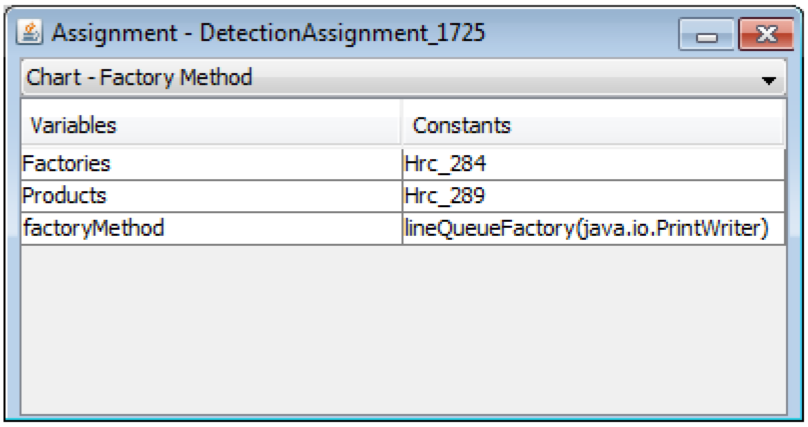
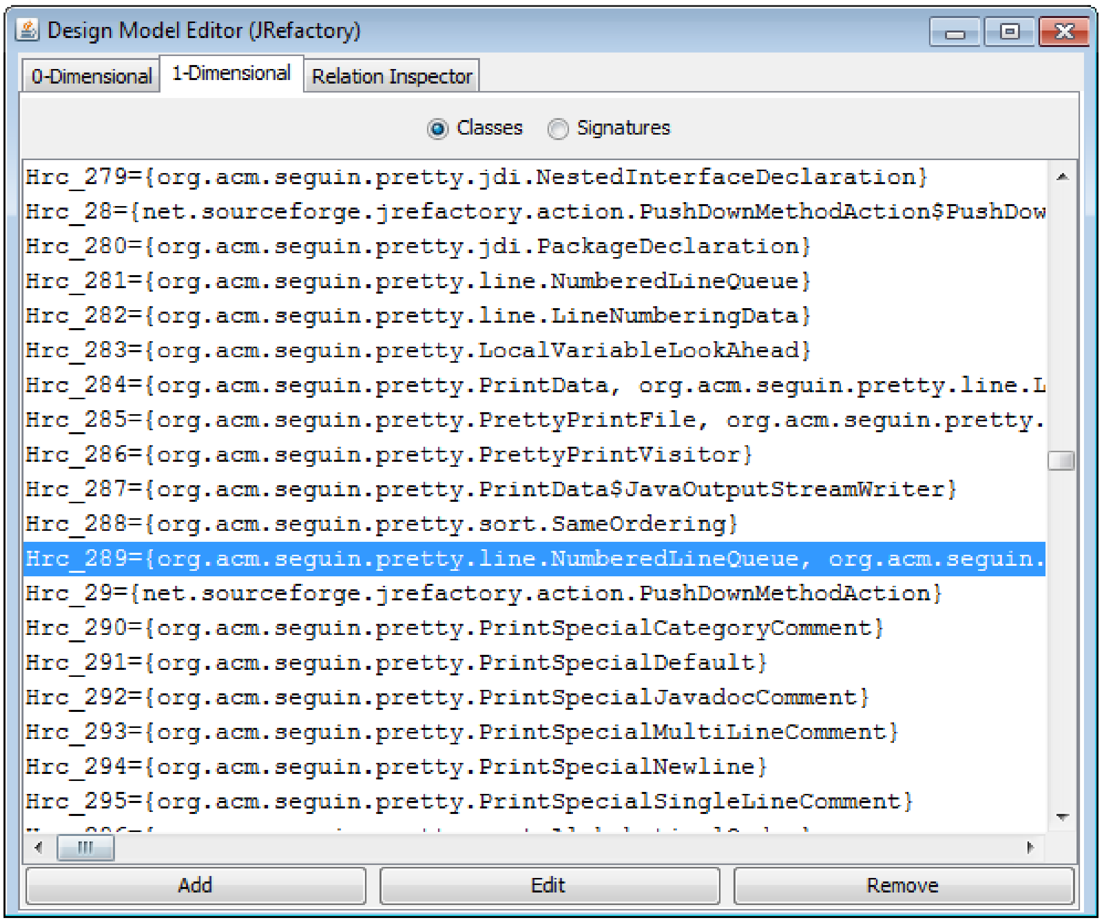


Figure 167: Codechart: An instance of Factory Method pattern detected in JRefactory



Variables	Constants
Factories	Hrc_284
Products	Hrc_289
factoryMethod	lineQueueFactory(java.io.PrintWriter)

Figure 168: an assignment of Factory Method detected in JRefactory



```
Hrc_279={org.acm.seguin.pretty.jdi.NestedInterfaceDeclaration}
Hrc_28={net.sourceforge.jrefactory.action.PushDownMethodAction$PushDow
Hrc_280={org.acm.seguin.pretty.jdi.PackageDeclaration}
Hrc_281={org.acm.seguin.pretty.line.NumberedLineQueue}
Hrc_282={org.acm.seguin.pretty.line.LineNumberingData}
Hrc_283={org.acm.seguin.pretty.LocalVariableLookAhead}
Hrc_284={org.acm.seguin.pretty.PrintData, org.acm.seguin.pretty.line.L
Hrc_285={org.acm.seguin.pretty.PrettyPrintFile, org.acm.seguin.pretty.
Hrc_286={org.acm.seguin.pretty.PrettyPrintVisitor}
Hrc_287={org.acm.seguin.pretty.PrintData$JavaOutputStreamWriter}
Hrc_288={org.acm.seguin.pretty.sort.SameOrdering}
Hrc_289={org.acm.seguin.pretty.line.NumberedLineQueue, org.acm.seguin.
Hrc_29={net.sourceforge.jrefactory.action.PushDownMethodAction}
Hrc_290={org.acm.seguin.pretty.PrintSpecialCategoryComment}
Hrc_291={org.acm.seguin.pretty.PrintSpecialDefault}
Hrc_292={org.acm.seguin.pretty.PrintSpecialJavadocComment}
Hrc_293={org.acm.seguin.pretty.PrintSpecialMultiLineComment}
Hrc_294={org.acm.seguin.pretty.PrintSpecialNewline}
Hrc_295={org.acm.seguin.pretty.PrintSpecialSingleLineComment}
```

Figure 169: Auto-defined higher-dimension entities (Hierarchies) (Factory Method - JRefactory)

Hierarchies details

Hrc_289={org.acm.seguin.pretty.line.NumberedLineQueue, org.acm.seguin.pretty.LineQueue}
 	
Hrc_284={org.acm.seguin.pretty.PrintData, org.acm.seguin.pretty.line.LineNumberingData}