MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Investigation of Violations in Software Design Patterns Implementations

DIPLOMA THESIS

Markéta Trachtová

Brno, 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Markéta Trachtová

Advisor: Bruno Rossi, PhD

Acknowledgement

I would like to thank the supervisor of my thesis, Bruno Rossi, for leading, professional advice and exceptional patience during the work on this thesis. I would also like to appreciate regular discussions about the problem under study. I am grateful for his provided time, wilingnes to help and kindness. I express special thanks to my family and colleagues for their understanding, and my freinds for supporting me in at all circumstances and basically making the writing of this thesis possible.

Abstract

The current thesis focuses on identification of design patterns violations in open source projects. As software decays over time, the original goals behind design implementation may be corrupted. The main aim, is to discuss identification of design pattern violations occurring in different projects and provide means for their measurement and evaluation of their relevance against definitions.

For identification and evaluation of patterns discovered within project, an approach using aspects of similarity characteristics occurring throughout all of design definitions is presented. For each pattern from catalogue by Gamma et al. [11] we create tables of identified characteristics and discuss their changes during implementation. We propose methodology for conformance measure based on modified Jaccard's coefficient and provide example of use in practical test. In the final stage of our work we discuss use of introduced approach for automatic testing.

Keywords

object orientated design patterns, design pattern violations, software decay, similarity measure, Jaccard's coeficient

Contents

1 Introduction

Software tools are accepted as part of our everyday live. Mobile applications are used for contacting friends or family, cars would not start, if their software does not allow it. Usually, nobody wonders about systems he uses, much less how complicated was their development and maintenance. What is of interest for everyone, can be covered by single word, that is functionality. The most successful systems are providing the most functions at the right time and form. Such systems are usually large, with thousands of lines of code and years of consecutive development in their live history. High complexity and size has its disadvantages, that is – the bigger the project is, the more opportunities for mistakes are present.

In a race for better quality achievement, developers came up with many ways of facilitating different supportive measures. One of those is incorporation of design patterns into code of application. Design patterns (DP) provide solution to common problems reoccurring over and over again in development of object oriented systems; they can impose structure to even the most complex of systems. Thus, ever since they made official introduction into the world of software engineering in most well-known catalogue of patterns [11] everybody can find them incorporated in thousands and thousands of systems. As projects, implementing patterns grow older, so do patterns within them. Consequently, with every bug fixed and new functionality added we can see design changing. Increase of coupling between pattern and non-pattern related classes, decay of physical and logical code structure or originally unintended addition of methods and attributes to inner structure of pattern code, can cause damage to the project [6]. Worsening of the system implementation over time is commonly referred to as decay and is recognised as one aspect of technical debt [22]. Although decay of software design causes severe problems to quality of whole project (e.g. increase of testability because of dependencies) its identification is a nontrivial matter. Root cause of problem is identification of violations per se. That is how to distinguish between code related to pattern realization and code that is harmful. Moreover we recognize, even amongst the non-pattern related code parts, which are violative and which are, on the other hand, indispensable for proper functionality of application.

The problem in question of this thesis is the identification of design pattern violations occurring in different projects, as well as measurement and evaluation of their relevance against definitions. Every pattern might be realized in various systems differently, as each problem has its own solution specifications. The programming language, used for creation of system, has its say in the matter as well (difference in syntax and semantics). Additionally, during software evolution patterns are subjected to deteriorative changes and decay. Therefore their inner, as well as outer structure changes greatly from the original, very often this is done unintentionally (increased coupling between classes due to forgotten methods etc.). Recognition of conformance for particular implementation instances towards its patterns definition, provides valuable insight on "health" of system under study and possible existence of violations within its source code, with no regards to the livetime stage in which the system currently is.

The aim of this thesis is to present an approach for identification and evaluation of violations in design pattern implementations. Our concept gains from existing similarities between design patterns, that is their characteristics, recognised as a key building structures not only by us, but other authors as well [18]. We define mentioned characteristics and present an evaluation methodology, which uses them for a measurement of

conformance coefficient of pattern realization and its respective definition. Several possibilities of violation scenarios occurring during similarity measurement are explored. We appraise their relevance for pattern decay. Consequently, to investigate possible function of suggested approach in practice a sampling test, over two open source projects JHotDraw 7.0.6 [1] and JUnit 4.12[2] was conducted. Last but not least potential use of violation identification and pattern conformance evaluation is discussed. Results from conformance measurement of pattern realizations towards definitions suggest, which patterns are more inclined towards violations than others. Likewise they can pinpoint parts of code, that have accumulated grime and are in need of refactoring in order to prevent spreading of decay any further.

1.1 Thesis outline

The structure of the current thesis is as follows: the first chapter provides motivation behind our work, introduces the problems discussed within the thesis and objectives we aimed to achieve. Chapter 2 contains an overview of related work concluded by other authors, which we used to broaden the understanding of domain under study. Chapter 3 holds the definition of our work's key terms and builds on top of the problem statement. Chapter 4 introduces the approach and methodology adopted to solve the problem. It explains all major steps for design pattern violations identification and conformance measurement. Chapter 5 includes the description of practical experiment conducted in order to test our approach on real open source projects and contains discussion of obtained results. Chapter 6 concludes with the study results, threats to validity and suggestions for future work.

1. JHotDraw `http://www.jhotdraw.org/`
2. JHotUnit `http://junit.org/`

2 Related work

Design patterns (DP) have been studied from various points of view by many authors. This chapter brings a research overview that focuses mostly on DP violations and their evaluation as this is the main focus of this thesis. It starts with possible design pattern definition styles and goes over studies targeting violations occurring during software evaluation: primarily researched by Izurieta alone [14] or in collaboration with different researchers Bieman [2], Schantz [17], Griffith [12], Dale [6]. Another area of interest for the current thesis is the research focused on design patterns scoring [18] and detection [19]. This research focuses on the detection of design patterns after they have been added to the source code. A relevant problem, as we think, is that in many projects developers are subject of turnover and knowledge of implemented patterns might be lost. In addition the size the complexity of current systems might make it more difficult to track instances of design patterns, that have been added over the time. Scoring of design patterns can be seen as a complementary area. The reason is that often - depending on the approach - each candidate pattern is given a score, based on the resemblance with the pattern definition. The way, in which the score is defined, is part of this area's interest. Which, in more general terms, can come down to measurement of software characteristics, looking to assign concrete numbers to real world characteristics of software artefacts and processes.

The famous Gang of Four (GoF) – Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides - published the first book on design patterns [11] in 1994. Since then, the book has been used countless times as a reference for studies on design patterns as well as guidance book for creation of better design and code. Although there are many ways to approach a definition - e.g. visual specification via three-model presentation of patterns by Lauder and Kent [15] or enrichment of UML models[1] with action semantics by Mak et al. [16] - the roots of the definition always go back to Gamma et al. This is also the resource that most of the people would refer to when discussing design patterns.

Despite the fact that definitions are the key element of our work, they are not the main topic in question. Our focus is on understanding the violations, that can occur when a design pattern is applied. In the GoF book, each pattern is defined by means of certain characteristics (e.g. class associations, inheritance), that define the way, in which a pattern should look like when implemented. The definition is not fully strict, in the sense that some characteristics might be included or not, according to the problem at hand, and the responsibilities that classes need to be assigned to. By violation, we mean the possible structural and conceptual differences that might be in place during the implementation of the design pattern. Why is this important? The answer is, that usually such changes accumulate over time. They represent one form of technical debt for the project similar to code decay. The implications of the violations are, that they might make successive changes to the software project more error-prone and more time-consuming. As such, having an instrument allowing detection of those violations (debts) can be useful to monitor the status of a software project for future changeability. There are many studies showing, that classes that are subject to structural violations, are more error-prone (e.g. [1]).

As the focus of this thesis lies on design pattern violations and their evaluation, this thesis reviews the early work of Clemente Izurieta and James Bieman on type of design

1. Unified Modeling Language `http://www.uml.org/`

pattern violations called decay, which occurs during patterns evolution [13]. In the study authors defined design pattern grime (code not participating on pattern realization, but still existing within pattern implementations) as a type of decay and divided the grime into three categories - class, modular and organizational grime. Consequently Izurieta in his doctoral dissertation studied the extent of software designs decay [14]. He recognised another type of design decay, aside from grime, called design pattern rot and noticed that this form of violations destroys structural integrity of patterns. Furthermore, results obtained by his research suggest, that grime build up has harmful effect on testability as it increases testing requirements. In addition, growth of pairings between classifiers participating on design realization is negatively affecting comprehensibility and adaptability of the patterns.

Later on, Izurieta cooperated with many researchers to obtain better understanding of design decay. Together with Schantz a taxonomy of modular grime is provided [17], while the taxonomy for class grime was created with help of Griffith [12]. In both cases apart from naming and definition of grime, effects of its particular types were tested. Whereas modular grime built up causes unintentional increase in coupling of DP classifiers, class grime has negative effect on understandability of design structure. Afterwards, Izurieta with Dale provided overview on impacts of design patterns decay on quality of project in [6].

In the end we focus on research concentrated on pattern evaluation. Firstly Strasser et al. in [18] address possibility of scoring patterns with use of the Role-Based Metamodeling Language (RBML) meta-model language [7] in combination with PlantUML[2] specification to calculate score of patterns conformance towards definitions. Downside of their work might be necessity of providing both RBML definition and PlantUml specification of any pattern, that should be evaluated. Secondly Tsantalis et al. aimed to create tool for detection of design patterns [19]. Within the study authors employ algorithm for measuring similarity between graph vertices as an instrument of pattern detection. However, affiniti scores received from measurements are not presented in paper, neither are they displayed to user of the tool. Reason for this lies in fact, that the purpose of application is to detect pattern instances present in the source code, not to evaluate correctness of their implementation.

Discussions about software decay and rot provided basis for definition of pattern violations used in the this thesis. The previous work on identification and evaluation of patterns revealed the possible perils of measurement and precise assessment of source code. In our work, we want to look at the operationalization of the collection of information about design patterns violations. That is, given a software project, can we have a global indicator of non-conformance? Given our discussion about technical debt, this could give information, about how much the project can have hidden future costs for maintenance and changeability. The same indicator could be used to compare different projects. To reach our goal, dessign pattern violations need to be defined and operationalized at the conceptual, but also in some physical format. Outcome of that process could allow the retrieval and storage of metadata information about patterns. This will make tool support easier in the future. The next chapter shows definition of design patterns violations by defining the concept, presents the problem statement and the objectives of the thesis work. All this information will allow, in subsequent chapters, to propose the approach followed. Included is the introduction of a scoring system to compare pattern instances from their supposed design.

2. PlantUML `http://plantuml.sourceforge.net/`

3 Design patterns violations

Software source code undergoes many changes through time. Modifications tend to occur more often in sections providing key functionality to a system such as design patterns [2]. Due to all the alterations, it is fairly easy to transform originally helpful DP implementation into an unusable, errors producing code, that is almost impossible to maintain. Evolution, however, does not happen overnight. Thus it would be ideal to detect violations in early stages of evolution and based on their severity and overall pattern performance decide to keep, refactor or discard them.

That is why the thesis is centered on the identification of violations against design pattern definitions at first. Secondly focuses on measurement of their impact on a code. In addition, presented is a way of comparing pattern definitions to implementation based on predefined characteristics and a possible scoring for discovered violations. Findings can be later used to create an automated tool, which would help to identify possibly malfunctioning or unusable design pattern realizations. Such an application might save time and resources during software maintenance and refactoring. We delve now into the necessary definitions and problem statement.

3.1 Pure design patterns

Initially, in order to work with design patterns, the definition style to work with had to be chosen. As mentioned in Chapter 2, there are several different possibilities how patterns could be described. Nevertheless, a starting point is always the same – Gang of Four design patterns catalog [11]. Gamma et al. specified design patterns with the help of class diagrams and texture descriptions. Since this seemed to be the most comprehensive and complex existing depiction, we decided to base of our study upon them as well.

GoF stated four key elements of every pattern definition:

Name – every design pattern part needs to have appropriate name. Finding it might prove to be unexpectedly hard task. GoF related naming to the pattern as a whole; from designers point of view. At first glance it naturally helps, even beginners, to orient in the pattern structure. Ultimately pure pattern implementation would not discard this effort and incorporated elements naming into its own structure.

Problem – patterns are only helpful if they are used on a correct place. Each was created to solve specific recurring situation. The way of implementation itself may be original for every project. On the other hand, the problem to be solved is always very similar. To try and use certain pattern to straighten a case that is not suited for it, usually leads to difficulties of which an unusable code is a minor one.

Solution – our main part of work is concentrated over this point. Gamma et al. gave us various design pattern templates supplemented with generous descriptions. Later part of the thesis goes over their notes and specifications with intention of creating scale of importance of specific pattern features for every introduced template.

Consequences – the last, but possibly the most important element of patterns is definition of their effects. How will they improve created project, how can they worsen it. Although this is not easily measurable item, everyone that is about to use design patterns in his or her code needs to be aware of this.

The thesis is concerned mostly with the solution element of whole definition. Reason for it lays in fact, that elements of name, problem and consequences are, by definition,

very vague so they would fit concrete recurring situations, but would not limit the usage only to them. Implementation possibilities are unlimited and those three elements are more explanatory than demanding to be strictly fulfilled. Therefore we can't clearly establish, whether they were satisfied or not.

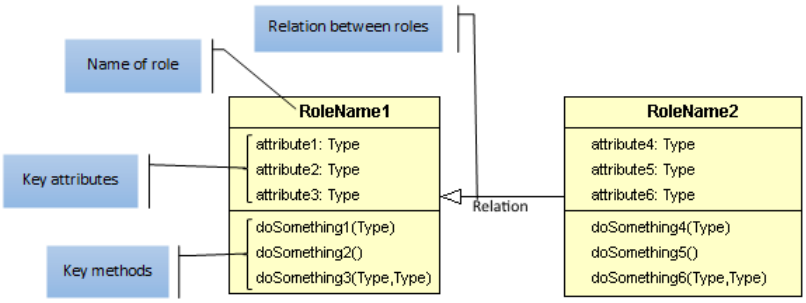


Figure 3.1: General role diagram

Within a context of the thesis, term pure design pattern definition is interchangeable with definition by GoF. In the scope of [11], solution elements are depicted with diagrams of roles similar to UML class diagrams (e.g. Figure 3.2) and complemented with written descriptions. Roles presented in definitions are equivalent to classes, interfaces or other classifiers participating on pattern realization. Their diagrams depict key attributes, methods and relations necessary for pattern functionality. Additionally, written comments explain in more detail purpose and relations between roles (Figure 3.1).

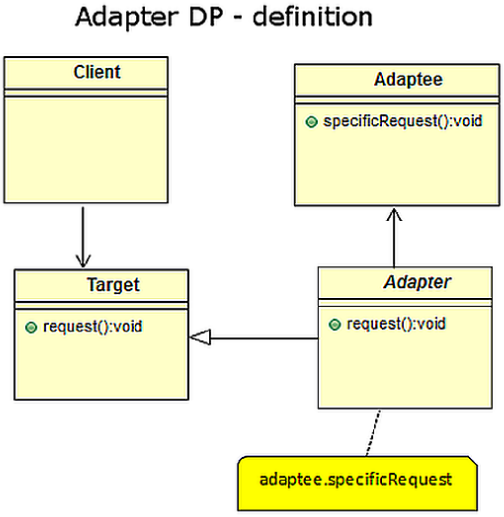


Figure 3.2: UML diagram of adapter design pattern from [11]

In parallel with the definition, there is a therm "pure design pattern implementation". Generally any realization of DP fulfilling all of requirements given by its pure design pattern definition and simultaneously being clear of any form of violations described in the following section could be called pure.

3.2 Design patterns violations

While definitions say clearly what has to be present in implementation, they are scarce on what must not. Code harmful to patterns performance may have many forms and effects. From organizational grime leading to chaotic and overloaded file structures to rot disintegrating DP structure integrity [6]. Nonetheless, violations do not necessarily have to be so severe. Neither are they always easily determinable.

Given how vague are definitions in regards to language specifications, we will not be discussing programming language related violations within scope of this thesis (e.g. differences in visibility scopes or relationships). Focus lays purely on general aspects going against definitions. Subsequently, for reasons mentioned in above section, attention fell mainly on solution element of definitions, thus only presence or absence of roles, attributes, methods, etc. is considered a violation. As a result, description of violations lies in comparison of implementations and their definitions.

For purpose of this work both sides of pattern – definition and implementation – are examined to see if violations are present. This is because the real projects have to provide complex functionality, that is carried out by attributes and methods, which do not always contribute to pattern realization. By mapping definitions against implementations, potential violations are located (Figure 3.3) and some of source code elements can be ruled out as non-harmful.

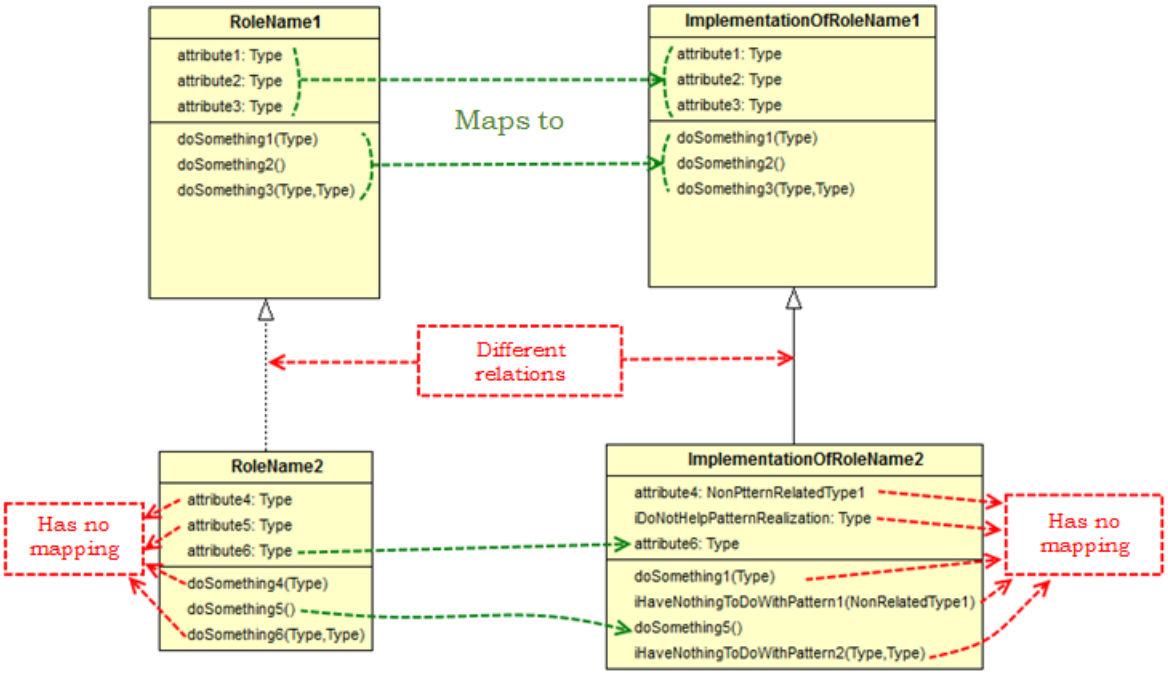


Figure 3.3: Role to implementation mapping

Situations occur, when it is questionable to say that discovered code is a violation. Although e.g. absence of role or presence of additional method within implementation could be strictly labeled as harmful. It may be completely different case when further context is considered. Here are some examples:

- Patterns are often interconnected and what seems to be a violation in one pattern,

can be core function for another since one class can substitute more roles from different patterns

- Pattern implementation does not always depend on all elements present in the definition, therefore the absence of some of them should not be considered a violation.
- Not all functionality is possible to obtain with pattern usage only, thus non-pattern related code is sometimes necessary to be present.

With regards to pure design patterns anything, aside from defined features, should be considered a violation. However, this strict rule would be very hard, if not impossible, to satisfy in real life. Additionally, as shown with examples above, precise separation of violations and pattern realization parts is not a trivial matter.

3.3 Statement of the problem

As mentioned in previous section, precise detection of harmful code is a complex problem. Subsequently, even if we are able to detect concrete violations for each pattern, unless it is possible to ensure that they are defections in global scope, the identification itself would not be useful. Therefore, rather than aspiring for specification of each and every single violation present in code separately, we aim to provide possibility of evaluation of pattern or project as a whole, counting in impact of harmful code as well.

Figure 3.3 in previous section introduces a general problem with comparing definitions and realization. Once this issue occurs in real projects, we can't rule out all elements without existing mapping as violations, since they might be relevant for complex functionality of project. Figure 3.4 shows similar situation to the one on Figure 3.3, only this time on the left we have concrete role definition and its counterpart, on the right side of picture, is taken from existing open source project. Let us assume, that all harmful elements would be discarded, or had to be refactored once identified. That would result into limitations in functionality and added massive amount of work to developers, in case of refactoring. Therefore while aspiring for comprehensive definition of project evaluation, it is crucial to address two major problems. Firstly there is a question of differentiation between possible violations and pattern parts of the code participating on pattern realization. Secondly there is the disclosure of an appropriate way of conformance measurement for both cases, as well as for the pattern and eventually project as a whole.

Complex evaluation of design patterns can have several benefits during development phases of project, as well as its support. While the system is under construction, structure is repeatedly changed. For that reason possibility of checking how much were patterns affected by adjustments might speed up detection of bugs or prevent more expensive changes in later phases of development. When considering maintenance, well kept code is always easier to work with and evaluation of patterns after new releases or bug fixes might reveal future drawbacks.

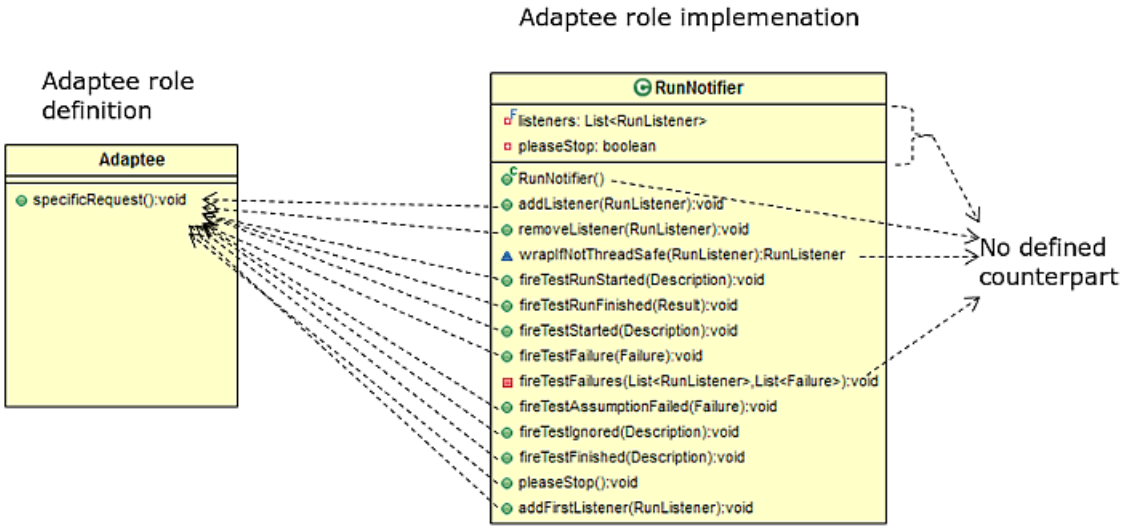


Figure 3.4: Concrete role to implementation mapping

3.4 Research objectives

With consideration to previous sections, objectives and sub-objectives of this thesis can be identified as follows:

Objective 1: Identification of violations against pure design pattern definition and their comparison to realization in existing open source projects.

- Sub-objective 1.1: Presentation and discussion of design patterns micro-characteristics.

Objective 2: Definition of scoring system for violations present in design pattern implementation.

- Sub-objective 2.1: Provision and discussion of a scoring system for design pattern violations.
- Sub-objective 2.2: Examination of score in practical experiment.

The above objectives were determined after evaluation of the problems introduced in this chapter. They cover all aspects related to DP violation identification and assessment. Respectively design pattern definition and implementation specifications. Last but not least, evaluation of possibly harmful code as well as future prospect of scoring tool is considered. Chapters 4 and 5 provide, respectively, the approach used for the resolution of the problem statement and the main results derived from the thesis.

4 Approach

In order to satisfy the objectives stated in the previous chapter methodology similar to the one used in [19] and [18] mentioned in Chapter 2 was adopted. These two studies have several traits corresponding to our work, such as comparison of definitions to implementations, scoring based on similarity or evaluation on real projects. For that purpose they served as a proven, inspirational templates on research approach. By combination of different methodology aspects from both papers we define five phases of approach towards the solution of the thesis (Figure 4.1). Phases go as follows:

- Phase 1 – definition of pure design patterns
- Phase 2 – identification of violations and micro-characteristics
- Phase 3 – transformation of obtained information into comparable values
- Phase 4 – definition of scoring of transformed values
- Phase 5 – experiment on real projects to validate the approach

Each phase provides explanation of the steps necessary for fulfillment of one or more of our goals. Particularly the identification of violations against pure pattern implementation is covered by first three phases and explained in more detail in section named "Pure design pattern micro-characteristics". Subsequently, the definition of a scoring system for violations present in design pattern implementations is addressed in phases three to five, which are discussed in the section "Scoring" and consecutively in Chapter 5. Sub-objectives are discussed, respectively, within the scope of the corresponding main objectives.

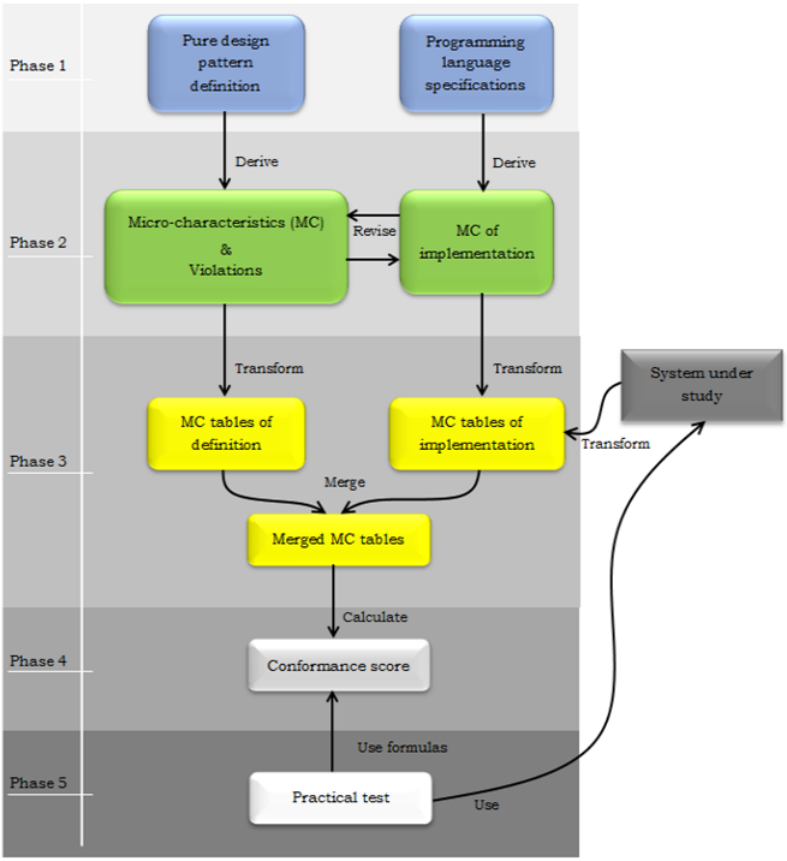


Figure 4.1: Summarization of the steps in the approach

4.1 Pure design pattern micro-characteristics

The first objective, that is identification of violations against design definitions, was addressed by the creation of templates based on micro-characteristics (MC) of pure patterns definitions. We arrange them with consideration of programming language specifications, which shaped the concrete implementation. Micro-characteristics are derived from reoccurring elements present in pure design pattern definitions proposed in Chapter 2 and subsequently enhanced with language specific requirements (e.g. visibility scope, types of classifiers, etc.). At this point (Figure 4.1 - Phase 1), the identification of programming language used for the thesis was fundamental, as particular characteristics of all pattern definitions are reliant on distinctive language features. Consecutively, with the use of role and class diagrams the obtained features were transformed into two sets of MC tables. One for definitions and the other for matching implementations (Figure 4.1 - Phase 3). Both types of tables are similar in structure with corresponding elements neatly narrowed down, which enables mutual comparison of their contents. Outcome of the correlation represents possible violations of pattern instance and can be used to assign score for the pattern conformity towards definitions. Before the steps for achieving conformance evaluation are revealed, we explain how the matrices of micro-characteristics are obtained. For this reason we have to recoil further back to initial pattern definition.

As mentioned in chapter 3, pure design pattern definitions are interchangeable with

the solution element of definition given by Gamma et al.. Their representation, provided in form of role diagrams is completed with written explanations on purpose of roles, relations and behavior. While going through different pattern definitions similar significant parts of design are repeatedly emerging in all descriptions. Identical observation was done by Tsantalis et al. in [19], where authors' detection algorithm profits from the fact, that within each pattern there is at least one inheritance hierarchy, which can serve as starting point for navigation between implemented roles. Unlike observations of Tsantalis et al., which were based on real life examples, recurring characteristics, identified in early stages of in this thesis, were derived only from definitions. Therefore the distinguished features were general enough to fit any object oriented language, yet not sufficiently specific to be used in evaluation. For purpose of obtaining characteristics comparable with patterns in real projects, which are implemented in one particular language, concrete conditions in conformance to given language (e.g.types of classifiers realizing the role, visibility and type of tributes, cardinality of relationships, etc.) have to be considered as well.

"The choice of programming language is important because it influences one's point of view." Gamma et al.[1]

When deciding on the language we consider several aspects, which had to be fulfilled by it. The first requirement is use of language in easily accessible open source projects. The reason is, that unless the language is common for system development, the structure and realization of patterns can't be evaluated, since source code of those project would not be accessible. Next criterion was existance of pattern definitions in given language, since prearranged definitions could be used for deepening of understanding of patterns, as well as for identification of language specific elements. Last aspect considered, was use of language in other studies targeting domain of design patterns. Results obtained by works based on same language chosen for purposes of this thesis, would consider corresponding specifications and could be more helpful to us than those built over different language characteristics, since language greatly affects realization of patterns.

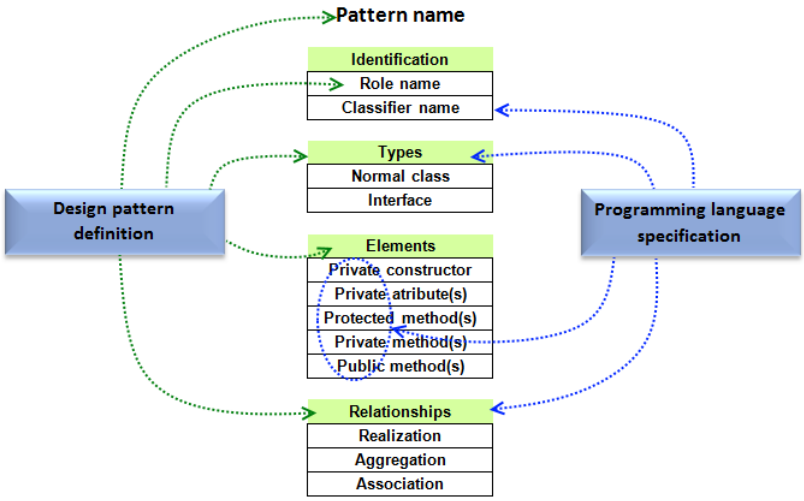


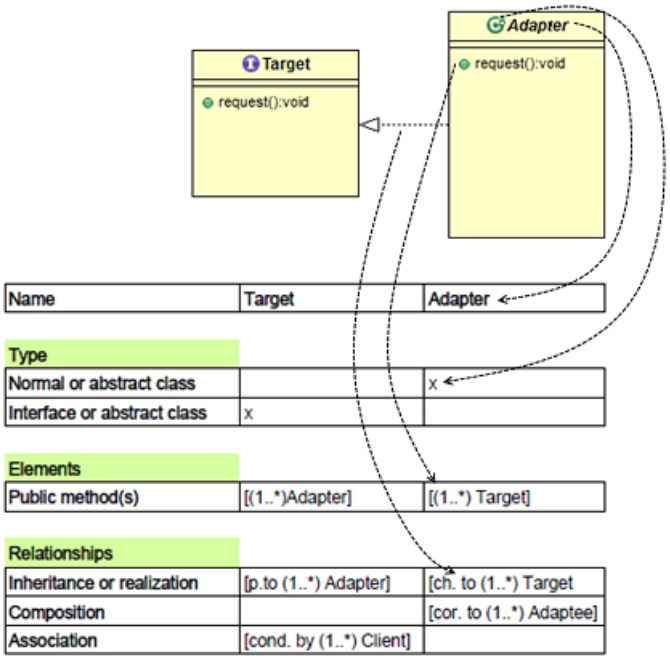
Figure 4.2: Origins of the elements

After consideration of the above aspects for different languages, we have decided to use the Java object oriented language. It is one of mainstream programming languages nowadays, thus there is fairly large amount of pattern definitions available. Consequently, finding open source projects with easily accessible source codes is not an issue. Last but not least, the judgement was affected by the fact that many of the studies from Chapter 2 are based on or, at least tested, on Java projects as well.

Subsequently, we combine language specific requirements and reoccurring elements derived from definitions into several sets of micro-characteristics (Figure 4.2). Each set contained features specific for one particular design pattern from catalogue in [11]. To achieve the easiest understandable form those sets were transformed into tables of micro-characteristics in following fashion.

First and foremost roles for each pattern definition, particularly their names and numbers, were noted. Roles represent a purpose of each class, interface or other classifier participating on pattern realization. They are therefore key elements of every pattern, thus serve as columns. Particular MCs of each role were put down as rows (Figure 4.3). As a result every table was divided into following four parts:

- Identification – name of role given by definition and its significance within pattern
- Type – states classifier type(s) which a role can acquire (e.g. normal class, interface, etc.)
- Elements – includes all kinds of attributes, constructors, methods, etc.
- Relations – covers relationships between roles



Name	Target	Adapter
Type		
Normal or abstract class		x
Interface or abstract class	x	
Elements		
Public method(s)	[(1..*)Adapter]	[(1..*) Target]
Relationships		
Inheritance or realization	[p.to (1..*) Adapter]	[ch. to (1..*) Target
Composition		[cor. to (1..*) Adaptee]
Association	[cond. by (1..*) Client]	

Figure 4.3: Mapping of elements to micro-characteristics

Figure 4.3 shows how micro-characteristics tables were derived from definitions on the example of Adapter and Target role of adapter pattern. Figure 4.4 provides, instead, the resulting product in shape of MC table of adapter pattern definition. Elements, within the tables have a simple true/false notation enriched with the representation of the amount of elements or related roles – if necessary. Blank cells stand for a false value, therefore they are present when the role has no characteristic of particular type, element or relationship. Character "x" or generally non-empty cells, depict the presence of a component. In the sections Identification and Type, the character "x" symbolizes the presence of particular characteristic. For the section Element, an existence of a characteristic is represented by a non-empty cell holding number of distinct elements in form of cardinality, taken over from UML notation. the part representing relationships has for each discovered characteristic the cardinality of the relationship and added notes of parents, children or containing classes for better understanding.

Name	Target	Adapter	Adaptee	Client
Core role		x	x	
Type "x" means that role has given type				
Normal class			x	x
Interface				
Normal or Abstract class		x		
Interface or Abstract class	x			
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*)Adapter]	[ch. to (1..*) Target	(1..*)	
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)				
Inheritance or Realization	[p.to (1..*) Adapter]	[ch. to (1..*) Target]		
[container to/contained by (Cardinality) Related role]				
Composition or Aggregation		[cor. to (1..*) Adaptee]	[cond. by (1..*) Adapter]	
Dependency				
Aggregation				
Association	[cond. by (1..*) Client]			[cor. to (1..*) Target]

Figure 4.4: Adapter micro-characteristics definition table

As stated above, Figure 4.4 reveals the complete matrix of characteristics required by an adapter pattern definition. From the table it is noticeable, that this particular pattern has four roles called Target, Adapter, Adaptee and Client. From them two are considered indisposable, namely Adapter and Adaptee. Targets classifiers type could determined as either an interface or an abstract class. The role contains at least one public method, that is implemented or inherited by its children - Adapter roles. Lastly, Target is in association relation with one or more Client roles of which he has no perception (Target role is on contained side of container – contained from direction). In similar fashion the rest of the roles could describe as well, thus ultimately in this manner the reconstruction of the whole role diagram would be possible. This approach only works for cases derived from role diagrams, since in this circumstances. As long as the characteristics are abided for type of roles and relationships; names of elements does not have to match original titles.

We are, however, using the tables for a completely different purpose, that is an identification of violations present in projects. The MC tables for pattern implementation instances were constructed on behalf of obtaining comparable counterparts to pure design definitions. Revision of assembling steps, performed for composition of definition tables takes us once again to definitions, namely to role diagrams. In order to create implementation equivalent of MC tables of definitions we go through all construction steps as if creating original matrices from square one. Only this time, the starting point is class of implementation instance instead of role diagram of definition.

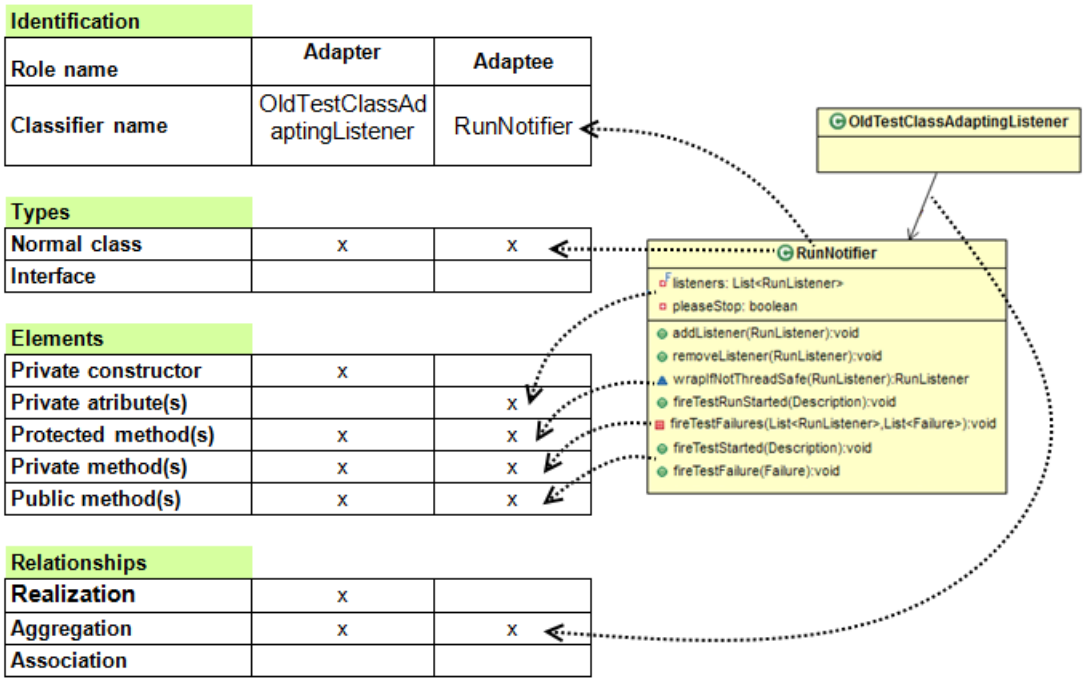


Figure 4.5: Transition from class diagram to MC implementation table

According to the expectations, the outcome of transformations is slightly different when starting from different diagrams. Firstly micro-characteristic tables of implementations have only true/false values. This feature is a result of aim to cover all elements of given class diagram. Assuming that the received diagram is a design pattern realization it is still desirable to capture all aspects of the implementation since it might contain violations. For the purpose of later comparison, the structure of tables is kept respectively to components of definition MC tables. Therefore, classifiers are given in columns, when types, elements and relationships are presented in rows. The result of the transition from class diagram (Figure 4.5) to micro-characteristic table of implementation is presented in Figure 4.6.

Name	Adaptee	Adapter	Target	Client
Core role	x	x		
Type				
Normal class	x	x		x
Interface			x	
Abstract class				
Elements				
Private Final atribute(s)		x		
Private constructor		x		
Private atribute(s)	x			x
Protected atribute(s)				x
Private method(s)	x	x		x
Protected method(s)	x			
Public method(s)	x	x	x	x
Relationships				
Realization (implemented)		x	x	
Aggregation	x	x		
Association			x	x

Figure 4.6: MC implementation table

Although data from Figure 4.6 might no longer be transformed back into the originating class diagrams, as it was possible for tables of definitions, the comparison of both types of matrices is a valid option now. Such assessment provides a set of possible violations as its result, which is exactly what was the original goal. Interested reader can find all the tables with our definitions in Appendi A. Exact course of actions, taken in order to acquire the resulting set, as well as scoring system concluded from actions is discussed in detail within next section.

4.2 Measurement and conformance scoring

"What is not measurable, make measurable."
Galileo Galilei (1564 - 1642)

In our everyday life we conduct measurements naturally, without even noticing. We estimate a time consumed by our trip to work, size of shoes we wear, price of items bought in a shop and many others. All in all, measurement provides better understandability of world around us. Fenton and Bieman [9] defined this mechanism formally in the following statement:

"Measurement is a process by which numbers or symbols are assigned to attributes of entities in real world in such a way so as to describe them accordingly to clearly defined rules."
Fenton and Bieman, 2014

When aspiring to narrow world phenomena into more visible and therefore controllable form we take existing entities, representations of complex objects (e.g a car or a design pattern) or events (e.g research for thesis or tea party), and identify their most prominent characteristics. An attribute is one of such aspects. It might be a feature as

well as property of the entity (e.g. color of the car or number of roles in design pattern). For the most part we want to obtain certain accuracy, thus the range is restricted or limits are put on types of measurement. Correspondingly, reduced, easily comprehensible and interpretable values (so-called indicators) are obtained. A clear interpretation and better understanding of the subject put under study is what we really want to gain from measurement. When standing alone, outcome of measurement is often indecisive. Therefore we strive to acquire a set of reduced values, rather than one single result. In the real world, this fact can be mirrored in many ways, before buying a car we take a look at a price of similar vehicles to determine, whether the cost of the one we picked is reasonable, when plucking apples the color of neighbor fruits is consider to pick the reddest one from the tree. Shortly, we gather and judge empirical results of measurements from our surrounding to interpret single value of subject under study correctly. Since there is certain kind of intelligence barrier present between changeful real world objects and strict numbers a longer road has to be taken to be able to obtain relevant values describing accurately what is in front of us (Figure 4.7).

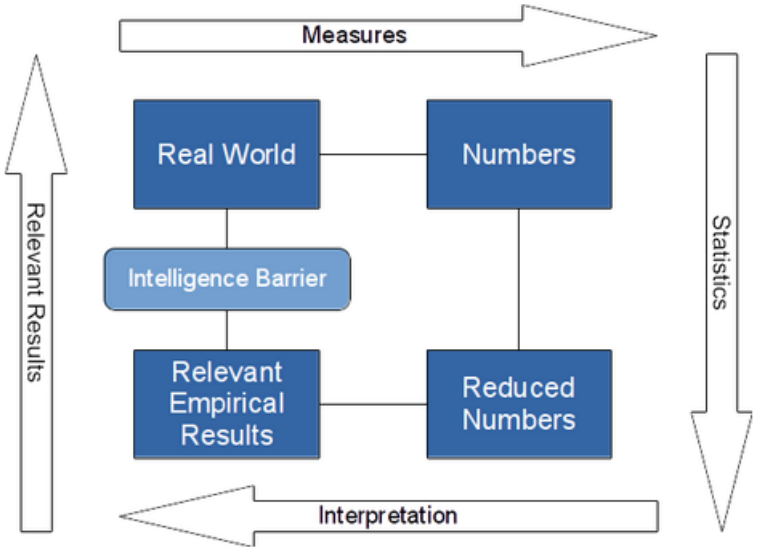


Figure 4.7: From real world to numbers [8]

This thesis, so far, provided entities in the form of design pattern definitions and their realizations. Recognized attributes to be measured are micro-characteristics represented as two sets of MC tables (Figure 4.1 - Phase 3). One set for definitions and the other for their implementation instances obtained from real open source project. Further on, numbers are assigned to those tables by firstly merging respective pairings together and secondly, by applying formulas selected carefully for purpose of obtaining conformance score of particular patterns. We bring together each pair of tables for pattern definition and its respective realization to prepare a ground for measurement of their conformance. For every characteristic occurring in one table or the other, the presence is noted in the merged matrix. In this fashion tables, containing only true/false representation of properties inherent to each specific pairing of MC tables, are created. For evaluation itself we adopted formulas based on types of values existing in merged matrices and desired interpretation. Since tables contain only true/false values we needed to explore

formulas concentrating on binary values. The purpose of measurement, that is obtaining a conformance coefficient for definitions and their realization, pointed our attention to similarity scoring. All in all, focused was centered on similarity measures over binary values. However those measures alone, would not be sufficient since different characteristics over all patterns have specific importance (e.g. core and optional roles). Therefore the similarity measure as well as formulas needed modifications in order to fit our needs. Our way of doing so, as well as the exact steps taken for merging tables are discussed in following part.

4.3 Merged characteristics

In order to conduct measurement existence of at least one concrete entity, with attributes to which we could assign values, was a necessity. As a result of transition from pure design pattern definitions and their correlating implementations instances into micro-characteristics and later into MC tables, several sets of complementing tables were obtained. Each MC table of pattern implementation (Figure 4.6) has its corresponding MC matrix of pure definition (Figure 4.4). This rule can be applied, with certainty, to realizations of patterns defined in the GoF's book. Further on, each pair of complementing matrices is merged into a single table representing aspects occurring in the definition as well as in its given implementation instance. While doing so, we noticed that only the limited scenarios depicted in Table 4.1 would apply.

Definition	Implementation	Explanation	Representation
yes	yes	the characteristic is present in MC table of definition as well as in MC table of implementation	11
yes	no	the characteristic is present in table of definition but is not in table of implementation	10
no	yes	the characteristic is not present in MC table of definition but can be found in MC table of implementation	01
no	no	the characteristic is not present in MC table and neither is in MC table of implementation	00

Table 4.1: Merging scenarios

From merging scenarios, two have a special meaning when related to merged tables. Firstly, the representation of the double negative "00" cases will never be explicitly depicted in merged, even though they are present in every empty cell of the table. Reason for it, is in the fact, that this scenario has no effect on the result of the comparison between patterns and their implementation. The awareness of absence of characteristic e.g. private constructor from implementation and also its non-existence in definition requirements, does not tell us anything about similarity of two sets of definition and implementation of design pattern. Thus its explicit representation from merged tables was disregard to achieve clarity and better readability of matrices.

The second notable case is the scenario represented as "01", when a particular trait (method, attribute, relationship, etc.) is present only in implementation with no prescription from the side of definition, it is a possible violation of pattern. On the other hand this feature might be highly necessary for providing key functionality for whole project. That is to say, code structure can be completely "healthy" or, with the same probability, deformative towards pattern design. Therefore we have equal chance, that the code under study is harmful or useful. This reality has to be taken into consideration while calculating conformance score later on.

Consecutively, merged tables consist of columns representing roles, respectively, classifiers, with characteristics are noted as two digit number on rows of particular features they provide. The numeric figure can only consist of digits 0 and 1. The first digit stands for absence or presence (0 or 1) of element in definition while second digit serves the same purpose only for implementation instance under study. Example of a complete merged table is shown in Figure 4.8.

Name	Adaptee	Adapter	Target	Client
Type				
Normal class	11	11		11
Interface			11	
Abstract class				
Elements				
Private Final atribute(s)		01		
Private constructor		01		
Private atribute(s)	01			01
Protected atribute(s)				01
Private method(s)	01	01		01
Protected method(s)	01			
Public method(s)	11	11	11	01
Relationships				
Realization (implemented)		11	11	
Aggregation	11	11		
Association			11	11

Figure 4.8: Merged table for MC of JUnit adapter pattern implementation

4.4 Scoring

For scoring itself, initially values representing measured attributes were taken into account. In case of merged tables it is binary enumeration, that is true/false. The implication of the purpose behind measurement, concretely conformance scoring, suggested suitability of the situation for use of similarity measurements. Similarity is generally viewed as likeness or resemblance of one object to another. In mathematics, particularly in geometry, two figures may vary in size and still be titled as similar, as long as they have same shape. The opposite of similarity, a distance or difference, stands for representation of amount of space between two places. We have considered several possible similarity/distance measures suggested by Cha[4]. In his study Cha presents over 40

similarity/distance measures divided into groups according to semantics, syntactic similarity, implementation admonitions and serving for comparison of nominal type histograms. This overview provided notional map of equations, from which we needed to choose those that could be more appropriate.

Eventually Jaccard's measure was determined to be the most appropriate one for purpose of this thesis. Firstly this measure is primarily focused on evaluation of similarity, secondly although it can be used for measurement of non-binary values, its power is demonstrated best when used on true/false values. However our opinion was most affected by one of key characteristics of Jaccard measure. Which is, its dismissal of cases when data are non-existent in both compared object. This trait perfectly corresponds with scenario presented in Table 4.1 - "00" which, as implied in above text, has no additional value to similarity/dissimilarity between definition and implementation.

For explanation of Jaccard's measure (alternatively also Jaccard similarity coefficient, a.k.a. Jaccard index) between two objects D and I, we have to firstly define three variables:

- ω_{11} – signifies number of variables where both objects are positive (corresponds with Table 4.1 representation "11")
- ω_{10} – signifies number of variables that are positive in object D and negative in object I (corresponds with Table 4.1 representation "10")
- ω_{01} – signifies number of variables that are negative in object D and positive in object I (corresponds with Table 4.1 representation "01")

Subsequently we can present measure formula S_{DI} as follows:

$$S_{DI} = \frac{\omega_{11}}{\omega_{11} + \omega_{10} + \omega_{01}} \tag{4.1}$$

Distance d_{DI} can, auxiliarily, be obtained with:

$$d_{DI} = 1 - \frac{\omega_{11}}{\omega_{11} + \omega_{10} + \omega_{01}} \quad thus \quad d_{DI} = \frac{\omega_{10} + \omega_{01}}{\omega_{11} + \omega_{10} + \omega_{01}} \tag{4.2}$$

The range of results can vary from 0 to 1 and represents the similarity/distance of two objects represented as asymmetric sets of samples. Jaccard's measure is used to determine similarity for each design pattern role separately before completing score for whole pattern. Examples of Jaccard's measures applied on general sets simulating scenarios of merging tables (Table 4.1), can be found in Figure 4.8: variations for variable $\omega_{11} < 1$ are not counted since result would be always 0. Also merging case "01" is divided in two versions a) and b). Version a) simulates case where all variables ω_{11}, ω_{10}, ω_{01} are present with non-zero values. Meaning that from the implementation one element is present in definition as well, one is not and one element required by definition is not implemented in role at all. Version b) represents the case, where variable ω_{10} has zero value. That is all elements required by definition are present in implementation, therefore case "10" has to have zero value. Since roles are key design elements, it is desirable to have their score for comparison, additionally individual patterns roles have different importance within pattern as some can be omitted and presence of others is mandatory (e.g. Concrete decorator role is optional in decorator pattern, whereas Component in the same design is obligatory). This reality is covered in complex scoring

	case"11"	case"10"	case"01" version a	case"00" version b
ω_{11}	1	1	1	1
ω_{10}	0	1	1	0
ω_{01}	0	0	1	1
Jaccard's coefficient	1	0.5	0.33	0.5

Table 4.2: Example calculation of Jaccard's coefficient with relation to merge scenarios

of each pattern instance. Prior obtaining conformance coefficient of whole instance roles need to be evaluated.

Four cases, possible to occur while merging MC tables were disclosed during the research. In accordance to that we suggest that two, out of those four, have special meaning. That is, representation labeled "00" and "01". Double negative is recognised as non-valuable information for similarity measure within this thesis, with no relation to a conformance scoring. Case "01", on the other hand holds important meaning for correct evaluation. Scenario representing absence of element in definition but presence of distinctive feature in implementation suggest, in equal measure, possible violation and code fundamental for project functionality thus gives us equal chance for identification of violation or mandatory attribute. That is in contrast to rest of the cases, where "11" and "10" are determining in identification of perfect match or violation due to missing element, the case "01" has half of the ability to disclose violations present in pattern. This decision is an example of the description of measurement theory: when going from the real world to the numerical world we need to make modelling decisions for the attributes in place.

The contribution of scenarios to the determination of the conformance score, can be different as well. When ω_{11} means complete satisfaction of requirements, the number ω_{10} suggests inconsistency. Concretely, $\omega_{10} \neq 0$ states the existence of definition requirements, that are not met by implementation. This fact suggests a possible violation and should lower the conformance score. Whereas ω_{10} = 0 demonstrates fulfillment of all conditions given by concrete definition and similarity score should be increased. In regards to examples in Table 4.2, the contribution of the different values in the scenarios is:

$$case\text{"11"} > case\text{"01"}\ variation\ b > case\text{"10"} > case\text{"01"}\ variation\ a\ > case\text{"00"} \quad (4.3)$$

It is apparent, from Table 4.2, that simple Jaccard's coefficient does not meet this new criterion since $case\text{"10"} = case\text{"01"} variationa$. To address the situation we modified Jaccard's measure to reflect amount of contribution of conformance measure by multiplication of number representing occurrence of case "01" by fixed constant $\frac{1}{2}$ to embody its lower deterministic ability. In order to keep the range of results between 0 and 1 we have to multiply the rest of variables in the formula by constant 2. Our new, modified similarity measure MS_{DI} between two objects D and I, based on Jaccard's index is as follows:

$$MS_{DI} = \frac{2 * \omega_{11}}{2 * (\omega_{11} + \omega_{10}) + \frac{1}{2} * \omega_{01}} \quad (4.4)$$

	case"11"	case"10"	case"01" version a	case"00" version b
ω_{11}	1	1	1	1
ω_{10}	0	1	1	0
ω_{01}	0	0	1	1
Jaccard's coefficient	1	0.5	0.33	0.5
Modified similarity coefficient	1	0.5	0.4	0.8

Table 4.3: Example calculation of modified similarity coefficient with relation to merge scenarios

The effect on merging scenarios is noticeable from Table 4.3. While the original Jaccard's measure could not reflect deterministic and contributional values of different cases, the modified formula can.

Excluded measurement of roles using the modified similarity formula, was later incorporated within the conformance measure of the whole pattern instance (Figure 4.10 - step 2). Similarity of pure pattern definition to its complete implementation instance could be calculated as a ratio between summarized resemblance scores of all roles and number of roles realizing given pattern implementation. However, likewise to the role similarity measurements, even for realization conformance the specific scenarios have to considered, that might occur in the middle of comparison. Since the number of roles within pattern acts as one of the key parameters, we focused on cases when the number of roles realizing the pattern is different than defined.

In the original GoF definition, some of roles are labeled as optional (e.g Pattern role of factory method pattern can be substituted with Concrete Pattern role with no harm done to design). Moreover, in many existing projects, it is very common to miss some of (by definition) non-optional roles as well. Additionally, design patterns identification tools often distinguish just key parts of every pattern (Client and Target role of adapter pattern are not shown by detection tool from [19]). Thus conformance measurement has to be applicable on patterns with more or less roles than states the definition.

The former case, when more roles are present in implementation, should by no means be considered harmful. Several pattern designs are built on a premise of easy expansibility (e.g for abstract factory pattern it is expected to have number of Concrete Factory and Product roles). The later case, omission of certain roles, has to be considered from perspective of core and non-core (optional) roles differently. Roles, by definition, optional could be missed from pattern implementation and this, in fact, should not affect conformance score of design. However, as mentioned above, practice has proven that not all non-optional roles are necessary for proper pattern realization. Example might be ConcreteDecorator role from decorator pattern which, although is not labeled by GoF as optional, can be omitted when substituted by Decorator itself. Concrete Decorator extends functionality of Decorator, therefore if rage of capabilities provided by Decorator itself is sufficient, role ConcreteDecorator does need to be implemented.

Contrary to that, core roles are indispensable, hence have to be present in pattern. Their absence would break down the pattern structure and consequently destroy the pattern. All things considered, this thesis is not about pattern identification. Therefore precondition for the work is that a code, received for study, have already been identified as a pattern implementation. Hence all parts necessary for pattern recognition are sure to be present, including core roles. Since presence and absence of non-core roles

provides ambiguous information about the pattern conformity, the possibility, which affects pattern conformance measure, was represented by multiplication of the score obtained from optional roles, with a fixed constant $\frac{1}{2}$. This constant speaks for an equal probability of violation occurrence inflicted by presence/absence of optional role. In order to obtain the similarity ratio corresponding with Jaccard's measure, that is result scale from 0 to 1, and simultaneously consider ambiguous nature of optional roles, the variable representing number of optional roles had to be scaled proportionally as well. As a result the range of conformance coefficient of pattern instance is <0,1>. Coefficient representing similarity for pattern P form modified similarity indexes of core roles CR and optional roles OR can be calculated according the formula:

$$P_{score} = \frac{\sum_{m=1}^{n} CR_m + \frac{1}{2} * \sum_{i=1}^{k} OR_i}{n + \frac{1}{2} * k} \tag{4.5}$$

Where:
P_{score} – similarity conformance score of whole pattern
n – number of core roles present in implementation
k – number of optional roles present in implementation
CR – modified similarity index of core role present in implementation
OR – modified similarity index of non-core(optional) role present in implementation

The real application of the above measurements on instance of adapter pattern found in system under study can be seen in Figur 4.9. Number of merging scenarios, occurring in each role are counted on respectively named row, from them following similarity coefficients of each role were derived as follows.

$$MS_{AR} = \frac{2 * 3}{2 * (3 + 0) + \frac{1}{2} * 3} = 0.80 \tag{4.6}$$

$$MS_{AO} = \frac{2 * 4}{2 * (4 + 0) + \frac{1}{2} * 3} = 0.84 \tag{4.7}$$

$$MS_{TT} = \frac{2 * 4}{2 * (4 + 0) + \frac{1}{2} * 0} = 1.00 \tag{4.8}$$

$$MS_{CT} = \frac{2 * 2}{2 * (2 + 0) + \frac{1}{2} * 4} = 0,67 \tag{4.9}$$

Where:
MS_{AR} – modified similarity of pair Adaptee role, RunNotifier classifier
MS_{AO} – modified similarity of pair Adapter role, OldTestClassAdaptingListener classifier
MS_{TT} – modified similarity of pair Target role, TestListener classifier
MS_{CT} – modified similarity of pair Client role, TestResult

Consequently results obtained from equations (4.6), (4.7), (4.8) and (4.9) are used to compute the similarity score of whole adapter pattern instance as:

$$P_{score} = \frac{MS_{AR} + MS_{AO} + \frac{1}{2} * (MS_{TT} + MS_{CT})}{2 + \frac{1}{2} * 2} \tag{4.10}$$

$$P_{score} = \frac{0.8 + 0.84 + \frac{1}{2} * (1 + 0.67)}{2 + \frac{1}{2} * 2} \tag{4.11}$$

$$P_{score} = 0.83 \tag{4.12}$$

The pattern conformance score, as shown in (4.12) states that instance of pattern in Figure 4.9 is on 83% similar to definition of adapter pattern. From the Figure 4.9 can be seen that there are no occurrences of "10" scenario which means, that all mandatory requirements from definition were met. Absence of "10" scenarios suggest that the only possible violations may be present within the code implemented beyond the elements prescribed by definition.

Name	Adaptee	Adapter	Target	Client
Type				
Normal class	11	11		11
Interface			11	
Abstract class				
Elements				
Private Final atribute(s)		01		
Private constructor		01		
Private atribute(s)	01			01
Protected atribute(s)				01
Private method(s)	01	01		01
Protected method(s)	01			
Public method(s)	11	11	11	01
Relationships				
Realization (implemented)		11	11	
Aggregation	11	11		
Association			11	11
Type Number of merging scenarios				
"11"	3	4	4	2
"10"	0	0	0	0
"01"	3	3	0	4
Moddified similarity index:	0.80	0.84	1.00	0.67

Figure 4.9: Adapter implementation comparison to definition

In similar fashion in which we retrieved the score with (4.12), we can evaluate all patterns in different projects. Aim of our work is to be able to define one system as a whole with regards to design patterns and their violations. Given the fact that we are capable of conformance estimation for separate patterns, we can consequently derive desired score for the whole project. Our similarity formula works with previously obtained pattern similarity scores. Since every project can be realized with the help of various patterns (e.g. singleton, adapter, factory method, etc.) respectively their many instances, we count conformance score for groups of instances affiliated with patterns of different types first and derive complete score for project later. Since we do not recognize any variations or special cases of pattern types we calculate their coefficient simply by the sum of conformance scores of individual instances divided by the number of instances of given pattern. Similarity measure T_{score} of pattern over all its instances is as follows:

$$T_{score} = \sum_{m=1}^{n} \frac{P_m}{n} \tag{4.13}$$

Where:
P_m – similarity score of pattern instance
n – number of patterns instances discovered within code

The score will represent the similarity of all instances of given pattern towards their definition. Such information might be interesting in disclosure of dissimilarity proneness of certain patterns over different projects, obtained information might suggest which patterns are generally better implemented than others. Range of results of formula (4.13) is kept between 0 and 1, which is given by the characteristics of equation. This range is easily comprehensible and applies as well for last equation of global project conformance.

Global project conformance measure represents "health" of all patterns implemented within it's scope. When calculating this score we summarize similarity scores of patterns measured over their instances and divide the sum by number of pattern types existing within project. The complex score for the whole project C_{score} would, consequently, be calculated according to the formula:

$$C_{score} = \sum_{i=1}^{n} \frac{T_m}{n} \tag{4.14}$$

Where:
n – number of patterns present in project
T_i – similarity score of all instance of concrete pattern

The similarity measure for whole project gives us valuable insight on a state of implemented patterns and amount of possible violations. The lower the score, the less accurately the tested patterns have been implemented. Low score might not necessarily mean wrong functionality or quality of the project, however, as observed by Izurieta, Bieman or Dale when patterns are imbalanced and overflowing with code not related to its realization testability, maintainability and comprehensibility of source code is heavily affected.

Throughout this chapter an explanation of exact steps how to get from definition of pure design pattern and introduction, as well as justification, of Java programing language as our base for implementations under study (Figure 4.1 – Phase 1), to formal definition of conformance measurement for project as a whole (Figure 4.1 – Phase 4). Definitions used are described in Chapter 3 and are interchangeable with those provided by GoF in [11] . Gamma et al. also noted, that choice of programming language affects a point of view for design pattern: our choice fell upon Java, since high amount of existing open source projects and pattern definition makes study of them easier.

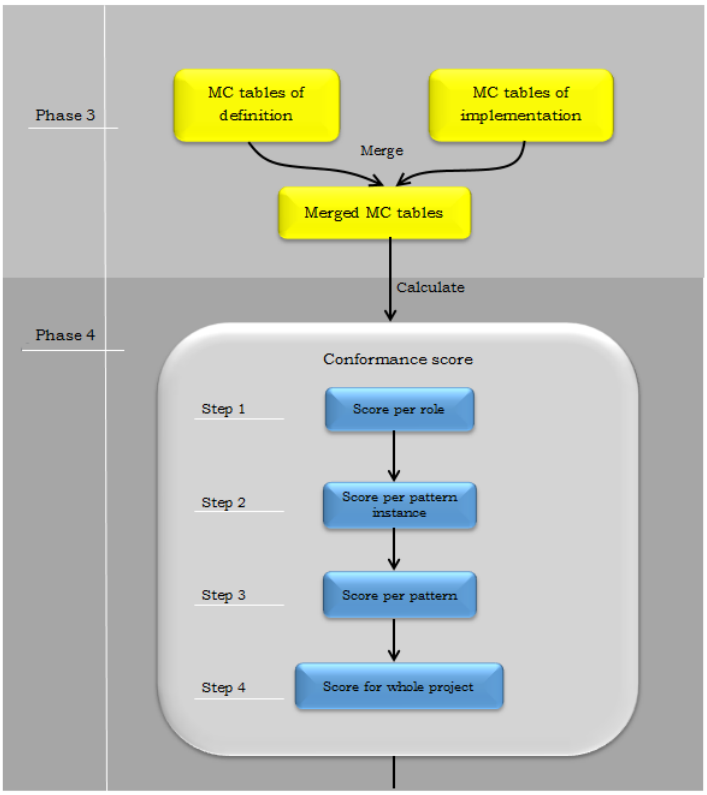


Figure 4.10: Measurement and scoring steps

From definitions, implementation and language specification we identified repeatable occurrence specific elements micro-characteristics (Figure 4.1 – Phase 2) and discussed their possible representation in form of MC tablesAs representation of features in form of tables is easily comprehensible, the description of transformations from characteristics into MC tables (Figure 4.1 and Figure 4.10 – Phase 3) was provided together with interpretation of values present in them. Tables were independently created for all 23 definitions given by GoF (included in appendices) and separately for implementations instances. After the realization was mapped to its definition specification MC implementation tables were created. As such comparable pairs of tables, MC tables of implementations and MC tables of definition, were available for further work. For purpose of better overview of differences between implementation and definition each pair of tables was into a single merged MC table. Such a table consists of binary, a.k.a. true/false, representation of aspects present in both original MC tables and is used as a stepping stone for conformance evaluation.

Similarity measure for whole project (Figute 4.1 Figure 4.10 – Phase 4) was concluded in several steps. Firstly the roles in merged MC tables were inspected to investigate how close are concrete realizations to their definitions (Figure 4.10 – step 1). At this stage several scenarios, that can occur when comparing two sets of true/false values, were taken into account. Subsequently, a measure called Jaccard's coefficient was slightly modified to be able to reflect those scenarios, when applied to measure conformance of roles. The modified similarity measure was used to proceed to another step, that is conformance measure of concrete pattern instance towards its definition (Figure 4.10 – step 2). A conformance scores of roles was received by modified measure, with its use a coefficient for concrete pattern realization was counted. Once again there are several

significant scenarios, that might occur in the measurement process, which have to be addressed during calculation of the coefficient (absence/ presence of optional and core roles). Results obtained by comparison of complete pattern to its definition signify how well the design pattern was applied or what amount of violations is present. The same way, in which one pattern implementation instance was processed , can be applied to process all the others present in a project. Subsequently we obtain a set of coefficients per pattern instances. For each pattern, successively, the collective score over all it instances present in project can be counted by adding up all instance scores into one value and dividing it by number of instances for concrete pattern (Figure 4.10 – step 3). Obtained results can be added up once again and divided by number of patterns present in whole project, which will return conformance score for project as a whole (Figure 4.10 – step 4).

Within the next chapter the practical use of our findings is introduce, as well as description of accomplishments of Sub-objective 2.2. On a practical test conducted on 20 pattern implementations in two open source project, our approach is demonstrated.

5 Practical experiment

To study how would conformance measurement, in approach defined within the thesis, do in real environment of open source projects. We have performed sampling test on two systems created with Java programming language. Concrete projects were chosen on the basis of initial survey during which we went over several, randomly picked, applications from open source project community SourceForge[1]. In order to see how much are concrete design patterns used in real systems, we test number of them for presence of design patterns. For detection of patterns we use tool introduced and created by work of Tsantalis et al. [19], which provides set of instances of every pattern present in the project. Additionally, to be able to compare systems between each other we need common unifying characteristic. For that purpose size of projects in lines of codes (LoC) was obtained. Measurement of the size was done with application called LocMetrics[2]. Figure 5.1 shows representative examples of selected systems, their size in lines of code (LoC) and number of detected instances of each pattern. On account of obtained data two projects, similar in size and pattern distribution were chosen for the test. Consequently, the conformance coefficients for roles, pattern instances and finally each project were calculated. The Obtained results from the experimentation are discussed at the end of this chapter.

	Project name								
	jhotdraw 7.0.6	Java LOIC	jUnit 4.12	Java Game Maker	GanttProject	GeoNetwork	FreeCol	jEdit	Art of Illusion
LoC	56697	1327	38804	5698	68380	156874	187451	187804	144266
Patterns instances									
Factory Method	9		6		1	1	1		1
Prototype	12					2	2		2
Singleton	8		7		3	19	19	1	18
(Object) Adapter - Comman	33		21	2	7	9		2	7
Composite	2		2						
Decorator	13		15			4	5		5
Observer	9		1						
State-Strategy	94	1	18	2	13	25	24	3	25
Tamplate Method	7	1	10		2				
Visitor	1								
Proxy			1						

Figure 5.1: Pattern usage overview

5.1 Experimental Units

Two open source project, created with the use of the Java programming language were used to test the approach described in this thesis. Concretely, JHotDraw 7.0.6 and JUnit 4.12 were examined. Part of the reason for this choice lies in the fact, that both systems are similar in size, with range between 30 and 60 thousands LoC. Moreover, both implement over 80 instances of different patterns. Second reason for their use is their difference of origin. While JHotDraw was created for educational purposes, JUnit was made with sole purpose to work as testing framework. To elaborate further JHotDraw is a Java based framework used for creating graphic editors. It is being released under LGPLv2[3]

1. SourceForge `http://sourceforge.net/`
2. LocMetrics `http://www.locmetrics.com/`
3. GNU Lesser General Public License, version 2.1 `https://www.gnu.org/licenses/lgpl-2.1.html`

licence and was originally made by Dirk Riehle for purpose of his dissertation on design patterns [20]. Similarly JUnit is a testing framework for Java programming language and is released under Eclipse Public License Version 1.0.[4] This system is largely used as it is one of xUnit testing familly in Java environment.

5.2 Methodology

Firstly we detect pattern instances with a use of [19]. For each pattern instance only core roles and significant methods are detected and presented by the tool. Therefore, to see complex pattern missing optional roles, if they are present, we have to search for them manually. Considering the amount of instances detected, just JHotDraw contains over 100 various pattern implementations, the test is conducted on 10 pattern instance samples per project.

Gamma et al. divided patterns into three sections - structural, behavioral and creational. For our choice of DP instances, we respected those sections and selected maximum of three patterns per each. Within sections, patterns with the most instances throughout projects were chosen. Consequently, from the creational patterns factory method was chosen, structural section is represented by adapter and decorator, while for behavioral category there is state pattern. The structural section has two patterns instead of one for the simple reason of "popularity" of adapter and decorator patterns. As shown in Figure 5.1, the adapter pattern is second in number of instances in JHotDraw and decorator is third. While in JUnit decorator keeps its third place in number of instances, adapter has the most instance within project. Therefore both of them earned a place in the test.

The distribution of instances picked from patterns goes as follows:

- creational pattern
  - factory method – 2 instances
- structural pattern
  - adapter – 3 instances
  - decorator – 2 instances
- behavioral pattern
  - state – 3 instances

For each instance, we created the MC tables of implementations and later merged them with the respective MC tables of definition. Afterwards, from the created matrices the similarity coefficient for every role in every instance was calculated with formula (4.5) (Table 5.1, Table 5.2). Within pattern instances, some optional roles were omitted in implementation which is in tables represented as N/A (not available).

4. Eclipse Public License, Version 1.0 (EPL-1.0) `http://opensource.org/licenses/eclipse-1.0.html`

JUnit score per role				
Adapter p.	**Adaptee**	**Adapter**	**Target**	**Client**
Inst. 1	0.80	0.84	1.00	0.67
Inst. 2	0.75	0.80	0.47	N/A
Inst. 3	0.60	0.67	1.00	1.00

Decorator p.	**Component**	**Decorator**	**Concr. Comp.**	**Concr. Comp.**	**Concr. Comp.**	**Concr. Comp.**
Inst. 1	0.21	0.52	0.53	0.50	0.53	0.53
Inst. 2	0.71	0.52	0.57	0.80	0.80	0.57

State p.	**State**	**Context**	**Concr.State**	**Concr.State**	**Concr.State**	**Concr.State**	**Concr.State**
Inst. 1	0.94	0.67	0.80	0.86	0.80	N/A	N/A
Inst. 2	1.00	0.60	0.67	0.75	0.67	0.75	0.75
Inst. 3	0.89	0.89	0.67	N/A	N/A	N/A	N/A

Factory Method p.	**Con. Product**	**Con. Product**	**Creator**	**Con. Creator**
Inst. 1	0.94	N/A	0.75	0.91
Inst. 2	1.00	0.47	0.71	N/A

Table 5.1: Role similarity coefficient for sample instances of JUnit

Adapter. p	**Adaptee**	**Adapter**	**Target**
Inst. 1	0.92	0.84	0.67
Inst. 2	0.80	0.89	0.75
Inst. 3	0.62	0.75	0.75

Deco-rator	**Com-nent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**	**Con.Com-ponent**
Inst. 1	0.71	0.48	0.50	0.24	N/A	N/A	N/A	N/A	N/A	N/A
Inst. 2	0.71	0.48	0.47	0.47	0.44	0.47	0.53	0.53	0.50	0.50

State p.	**State**	**Context**	**Concrete State**	**Concrete State**
Inst. 1	0.75	0.86	N/A	N/A
Inst. 2	1.00	0.80	0.75	N/A
Inst. 3	1.00	0.80	0.57	0.80

Factory Method p.	**Product**	**Con. Product**	**Creator**	**Con. Creator**	**Con. Creator**
Inst. 1	0.92	0.50	1.00	0.38	N/A
Inst. 2	N/A	0.53	0.75	0.67	0.63

Table 5.2: Role similarity coefficient for sample instances of JHotdraw

Afterwards, the score per pattern instance was generated according to formula (4.5)(Table 5.3). From the results in tables 8,9 and 10 it is evident that Adapter and State pattern are generally better implemented than Decorator and Factory method. The former two patterns have score above 0.7 in all instances, the later patterns score dropped on 0.5 and even 0.4 at one instance. Such a low score indicates presence of "10" cases in pattern implementation. That is the instances of decorator pattern are missing elements required by definition.

	Adapter			**Decorator**		**State**			**Fact. Method**	
Score per instance	Inst. 1	Inst. 2	Inst. 3	Inst. 1	Inst. 2	Inst. 1	Inst. 2	Inst. 3	Inst. 1	Inst. 2
JHotDraw	0.84	0.83	0.70	0.52	0.52	0.80	0.87	0.83	0.76	0.67
JUnit	0.83	0.71	0.76	0.45	0.65	0.82	0.75	0.84	0.84	0.66

Table 5.3: Pattern instance similarity coefficient for samples from tested systems

Since for testing only samples of pattern instance were used - not whole set present in implementation - the complex conformance score calculated for the whole project is therefore only informational and inaccurate. The rest of pattern instances might change the outcome significantly. The project conformance scores from samples was obtained with use of formula (13) as follows.

$$JUnit_{score} = \frac{0.79 + 0.52 + 0.83 + 0.72}{4} = 0.71 \tag{5.1}$$

$$JHotDraw_{score} = \frac{0.77 + 0.55 + 0.80 + 0.75}{4} = 0.72 \tag{5.2}$$

Where:
$JUnit_{score}$ – conformance score of project JUnit
$JHotDraw_{score}$ – conformance score of project JHotDraw

The complex conformance scores for both projects are close to each other, in numbers 0.71 for JUnit and 0.72 for JHotDraw. This indicates that state of pattern implementations in both systems is currently on similar level. Educational origins of JHotDraw, targeting design patterns, suggest that originally quality of pattern implementation in this project was better than in JUnit. However in its 10 years of evolution and over 7 official stable releases decayed affected pattern quality negatively. Therefore now it is in state similar to JUnit, which being on it's 4th release status, provided less opportunities for injection of violations.

5.3 Results

We used the approach described within the thesis on two open source projects JHotDraw 7.0.6 and JUnit 4.12 to calculate their complex conformance score. Since there is over 100 pattern instances present in those systems and the test is done with limited tool support, we selected 20 representative samples from patterns with the most instances. From those samples we calculated the score per role, consequently per instance and pattern to finally obtain a complex score for both projects.

Results show that from all the sample instances the best outcome have Adapter and State patterns, whereas Decorator and Factory Method suffer from more possible violations. Lower score for both Decorator and Factory Method indicates absence of elements required form definition in their realizations. Although the complete score for projects is affected by the presence of Decorator pattern coefficients 0.71 for JUnit and 0.72 for JHotDraw are rather high in range (0.0,1.0). However the fact that only samples of patterns were used for the conformance testing needs to be considered. The drop in score, caused by Decorator, might be repeated by other patterns, included in this test. Similarly, the score might benefit from other well implemented patterns as well. In order to give a conclusive conformance score for whole system, all instances of all patterns would have to be tested. Furthermore only two projects were subjected to testing, in order to provide relevant results of empirical test over more projects should be done.

Validity threats for this thesis are further explained in next chapter together with conclusion of our work and possibilities of future actions.

6 Conclusion

The identification of design pattern violations within open source systems is a complex task. Reliable tools, that would target those violations are yet to be developed also for the theoretical implications of the definition of violations in implementation. The major contribution of this thesis to the domain of design patterns, includes an approach for conformance measurement of implemented designs towards their definitions. To conclude our study, we discuss the threats to validity of the research, the fulfilment of our objectives and future works.

6.1 Threats to validity

The validity of a study represents the level of thrust, that can be assigned to results according to the approaches used by researchers [3], [21]. Given the multiple facets of a study, there are many aspects, that need to be taken into account in terms of validity. As such, we report threats to validity following the schema in [3] that distinguishes among construct validity, internal validity, external validity and reliability. Regarding these different aspects, we will just explain them briefly to simplify the understandability of the subsequent discussion.

Construct validity relates to the way, in which the measures set in place are representative of the overall construct, that was set-up from a theoretical point of view. It can be easy to set-up a measure and have a valid data collection procedure, but it might be that it does not represent the original idea from which the study derived.

Internal validity deals with causal relations, that is how much can we assign the cause of a change in a variable of a second variable while the results might, in fact, be dependent from a third (hidden) variable. So we might give as explanation for a whole phenomenon one variable, that is not really important. Relates to this - and somehow funny - there is the whole world of "non-valid" correlations, e.g. one might assume that "the lack of pirates is causing global warming" (see [10]) - but this introduces another aspect, in that correlation does not imply causality, in the funny example, is the lack of pirates to cause global warming or global warming causing the lack of pirates?

External validity deals with how much easy is to generalize the findings to other cases. So the point here is on looking how much the final results could apply to similar cases. Usually, as the context is important it can be difficult to generalize over many cases, or at least there needs to be large statistical evidence to provide statistical significance for generalizability.

Reliability is concerned with how much the data analysis and procedure is dependent upon the specific researcher. If another person follows the same approach, will he get the same results or there are some validity threats in this sense?

To note that there also additional, more fine grained categories for threats to validity reporting (e.g. [5] that contains aspects such as maturity, that is how much the maturation of subjects could influence the results of a study analysis). We believe that the classification in [3] is the most interesting for the current study.

Discussing the threats to validity of the current thesis work, we can report the following. The aim of determining violations of design patterns implementations is dependent upon the way, in which the patterns have been characterized. Our definition of the patterns is based on the micro-characteristics, that we identified from theory: they

way in which these have been operationalized in terms of binary features and later with a scoring system are clearly part of construct validity. In our opinion, the translation of violations to the numerical world represents our understanding of the theory, but different representations can give different results in this sense. More experiments will be needed once some automated tool support will be available so that different scoring systems and representations could be evaluated.

Related to *internal validity*, the current study was not much investigating causal relationships among variables, and considerations made on the set of projects sampled are purely observational. We do not draw any conclusion about the compliance of projects to design patterns definition, but rather we use them to evaluate the proposed scoring system.

Related to *external validity*, the current thesis is only limited to the GoF patterns, furthermore all the work has been done by using as reference Java implementations of design patterns and on a limit set of sampled patterns from - again - some sampled projects. Also in this sense, automation by tooling could help in improving generalizability of the results.

Related to *reliability*, the procedure explained in the thesis should help other independent researcher to follow through the steps and replicate the results in the most compliant way. Definition of the pattern templates or instances should help others to refer to them to start independent usage. As per other threats to validity, also here a great help can come from the automation of the process. At that point the threats of validity will depend on how much the tools implement correctly the steps (and additional requirements) that have been characterized in the current thesis.

6.2 Research contribution

The evaluation of similarity between patterns and definitions provides valuable insight to the state of design implementation. As a project evolves design patterns exhibit a tendency to decay. With our approach the measurement of violations withi design patterns can be determined. Consequently, parts with high amount of violations are located, which provides information for possible refactoring to software development teams.

In this research, we used designs from the famous GoF catalogue to define pure design patterns. From the proposed definitions, recurring micro-characteristics were derived to create MC tables of definitions. We choose the Java programming language as basis of our study for reasons explained in Chapter 4 and incorporate its specification into MC tables. Result, in form of 23 tables with definition micro-characteristics can be found in appendix A. Similar tables were later constructed also for pattern instances from systems under study. Subsequently, MC tables of implementations were merged with those of definitions to create merged MC tables. Creation of those tables is important as they are used for similarity measurement. A modified Jaccard's coefficient was introduced to measure conformance of classifiers representing pattern roles towards the definitions. Likewise formulas, for similarity measurement of patterns and later of whole project were provided. We discussed the possible use of the introduced approach and its consequences in a practical experiment conducted on 20 pattern samples from two open source projects JHotDraw 7.0.6 and JUnit 4.12. Finally, we discussed the threats to validity of our study and provide suggestion for future use of our approach in automated testing. The objective of this thesis were fulfilled with contributions of our

work as follows:

- Objective 1
  - Sub-objective 1.1.
    - Pure design pattern definition.
    - Set of 23 tables with micro-characteristics of each object oriented design pattern from GoF catalogue.
- Objective 2
  - Sub-objective 2.1.
    - Approach for conformance measure of pattern instance and project as a whole.
    - Modified Jaccard's coefficient for similarity measure of roles and their realization.
  - Sub-objective 2.2.
    - Practical experiment and discussion of results.

6.3 Future work

Regarding related works, the first and most important would be to automate what has been described in the current thesis. The idea is to automate the process of analysis so that the violations could be computed automatically. Ideally, tool support should help already from the data collection phase, so that projects data could be automatically collected based on repository information, to run the analysis on large number of projects.

Bibliography

[1] ARISHOLM E., BRIAND L. C.. *Predicting Fault-prone Components in a Java Legacy System.* In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, New York, NY, USA, 2006, pp. 8–17.

[2] BIEMAN James M., STRAW Greg, WANG Huxia, WILLARD MUNGER P., ALEXANDER Roger T.. *Design Patterns and Change Proneness: An Examination of Five Evolving Systems.f,* n.d. doi: http://www.cs.colostate.edu/~bieman/Pubs/DPCmetrics03.pdf.

[3] CLAES Wohlin, RUNESON Per, HOST Martin, OHLSSON Magnus C., REGNELL Bjorn , and WESSLEN Anders . *Experimentation in software engineering*. Springer Science Business Media, 2012.

[4] doi: http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.154.8446

[5] COOK, Thomas D., CAMPBELL Donald Thomas , and ARLES Day. *Quasi-experimentation: Design and analysis issues for field settings*. Vol. 351. Boston: Houghton Mifflin, 1979.

[6] DALE Melissa R. and IZURIETA Clement. *Impacts of Design Pattern Decay on System Quality*. In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 37:1–37:4. ESEM '14. New York, NY, USA: ACM, 2014. doi: 10.1145/2652524.2652560.

[7] DAE-KYOO Kim, FRANCE R., GHOSH S., and SONG E.. *Using Role-Based Modeling Language (RBML) to Characterize Model Families.* In Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings, 107–16, 2002. doi: 10.1109/ICECCS.2002.1181503.

[8] EBERTC. and DUMKE R. , *Software Measurement: Establish - Extract - Evaluate - Execute*, Softcover reprint of hardcover 1st ed. 2007 edition. Springer, 2010.

[9] FENTON, Norman, and BIEMAN James . *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, 2014.

[10] http://www.forbes.com/fdc/welcome_mjx.shtml

[11] GAMMA Erich, HELM Richard, JOHNSON Ralph, VLISSIDES John. *Design Patterns: Elements of Reusable Object – Oriented Software. : Pearson Education*, 1994. ISBN 0321700694.

[12] GRIFFITH Isaac, IZURIETA Clemente. *Design Pattern Decay: The Case for Class Grime.* In Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 39:1–39:4. ESEM '14. New York, NY, USA: ACM, 2014. doi: 10.1145/2652524.2652570.

[13] IZURIETA C., BIEMAN J.M.. *How Software Designs Decay: A Pilot Study of Pattern Evolution.* In First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007, 449–51, 2007. doi: 10.1109/ESEM.2007.55.

[14] IZURIETA Clemente. *Decay and Grime Buildup in Evolving Object Oriented Design Patterns*. Colorado State University, 2009.

[15] LAUDER A., KENT S.. *Precise visual specification of design patterns.* In: Lecture Notes in Computer Science. vol. 1445. ECOOP'98, Springer, 1998, pp. 114–134

[16] MAK Jeffrey K. H., CHOY Clifford S. T., and LUN Daniel P. K.. *Precise Modeling of Design Patterns in UML.* In Proceedings of the 26th International Conference on Software Engineering, 252–61. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004. http://dl.acm.org/citation.cfm?id=998675.999430.

[17] SCHANZ Travis, IZURIETA Clemente. *Object Oriented Design Pattern Decay: A Taxonomy.* In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, 7:1–7:8. ESEM '10. New York, NY, USA: ACM, 2010. doi: 10.1145/1852786.1852796.

[18] STRASSER S., FREDERICKSON C., FENGER K. and IZURIETA C.. *An Automated Software Tool for Validating Design Patterns*. Honolulu, 2011. doi: http://www.cs.montana.edu/~shane.strasser/papers/an_automated_software_tool_for_validating_design_patterns.pdf.

[19] TSANTALIS N., CHATZIGEORGIOU A., STEPHANIDES G., and HALKIDIS S.T.. *Design Pattern Detection Using Similarity Scoring.* IEEE Transactions on Software Engineering 32, no. 11 (November 2006): 896–909. doi: 10.1109/TSE.2006.112.

[20] RIEHLE, Dirk. *Framework Design: A Role Modeling Approach*, 2000. doi: http://dirkriehle.com/computer-science/research/dissertation/index.html.

[21] YIN, Robert K. *Case study research: Design and methods*. Sage publications, 2013.

[22] ZAZWORKA Nico, VETRO Antzonio, IZURIETA Clemente, WONG Sunny, CAI Yuanfang , SEAMAN Carolyn and SHULL Forrest . *Comparing Four Approaches for Technical Debt Identification*. Software Quality Journal 22, no. 3 (September 2014): 403–26. doi: 10.1007/s11219-013-9200-8.

A Appendix - MC Tables of definitions

From here on we present micro-characteristics tables of definitions. In following order:
Table A.1: Absctract Factory
Table A.2: Singleton
Table A.3: Builder
Table A.4: Factory Method
Table A.5: Prototype
Table A.6: Proxy
Table A.7: Facade
Table A.8: Flyweight
Table A.9: Class Adapter (inheritance)
Table A.10: Object Adapter (composition)
Table A.11: Bridge
Table A.12: Composite
Table A.13: Decorator
Table A.14: Chain of Responsibility
Table A.15: Command
Table A.16: Interpreter
Table A.17: Iterator
Table A.18: Mediator
Table A.19: Memento
Table A.20: Observer
Table A.21: State
Table A.22: Strategy
Table A.23: Template Method
Table A.24: Visitor

Table A.1: Abstract Factory

Identification					
Name	Abstract Factory	Concrete Factory	Abstract Product	Product	Client (optional)
Core role	x			x	
Type "x" means that role has given type					
Normal class		x		x	x
Interface					
Abstract class					
Interface or Abstract class	x		x		
Elements [(Number of elements) role to share the elements with if there is any]					
Private constructor					
Static (final) atribute					
Public method(s)	[(1..*) Concr. F.]	[(1..*) Abst. F.]	[(1..*) Product]	[(1..*) Abst. P.]	
Relationships [parent/child to (Cardinality) Related role]					
Inheritence (extended)					
Realization (implemented)					
Inheritance or Realization	[p. to (1..*) Concrete F.]	[ch. to (1..*) Abst. F.]	[ch. to (1..*) Product]	[p. to (1..*) Abst. P.]	
[container to/contained by (Cardinality) Related role]					
Aggregation					
Association	[cond. by (1..*) Client]	[cor. to (1..*) Product]	[cond. by (1..*) Client]	[cond. by (1..*) Concr. F.]	[cor. to (1..*) Abst. F.] [cor. to (1..*) Abst. P.]

Table A.2: Singleton

Name	Singleton
Core role	x
Type "x" means that role has given type	
Normal class	x
Interface	
Abstract class	
Interface or Abstract class	
Elements [(Number of elements) role to share the elements with if there is any]	
Private constructor	x
Static (final) atribute	x
Public method(s)	(0..*)
Relationships [parent/child to (Cardinality) Related role]	
Aggregation	
Association	[(1..1) Singleton]

Table A.3: Builder

Name	Builder	Concrete Builder	Director	Product
Core role	x		x	
Type "x" means that role has given type				
Normal class		x	x	x
Interface				
Abstract class				
Interface or Abstract class	x			
Elements [(Number of elements) role to share the elements with if there is any]				
Private constructor				
Static (final) atribute				
Public method(s)	[(1..*) Concr. B.]	[(1..*) Builder]	(1..*)	
Relationships [parent/child to (Cardinality) Related role]				
Inheritence (extended)				
Realization (implemented)				
Inheritance or Realization	[p. to (1..*) Concr. B.]	[ch. to (1..*) Builder]		
[container to/contained by (Cardinality) Related role]				
Aggregation	[cond. by (1..*) Director]		[cor. to (1..*) Builder]	
Association		[cor. to (1..*) Product]		[cond. by (1..*) Concr. B.]

Table A.4: Factory Method

Name	Product (Optional)	Concrete Product	Creater (Factory)	Concrete Creator
Core role			x	
Type "x" means that role has given type				
Normal or Abstract class		x		x
Interface				
Abstract class				
Interface or Abstract class	x		x	
Elements [(Number of elements) role to share the elements with if there is any]				
Private constructor				
Static (Final) atribute				
Public method(s)			[(1..*) Concr. F.]	[(1..*) Creator]
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)				
Inheritance or Realization	[p. to (1..*) Concr. P.]	[ch.to (1..*) Product]	[p.to (1..*) Concr. F.]	[ch. to (1..*) Creator]
[container to/contained by (Cardinality) Related role]				
Aggregation				
Association	[cond.by (1..*) Creator or Concr. Creator]	[cond.by (1..*) Creator or Concr. Creator]	[cor. to (1..*) Product or Concr. P.]	[cor. to (1..*) Product or Concr. P.]

Table A.5: Prototype

Name	Client	Prototype	Concrete Prototype
Core role	x	x	
Type "x" means that role has given type			
Normal or Abstract class	x		x
Interface		x	
Abstract class			
Interface or Abstract class			
Elements [(Number of elements) role to share the elements with if there is any]			
Private constructor			
Static (final) atribute			
Public method(s)	(1..*)	(1..*)	[(1..*) Prototype]
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)		[p. to (1..*) Concr. P.]	[ch. to (1..*) Prototype]
Inheritance or Realization			
[container to/contained by (Cardinality) Related role]			
Aggregation			
Association or Composition	[cor. to (1..*) Prototype]	[cond. by (1..*) Client]	

Table A.6: Proxy

Name	Subject	Real Subject	Proxy
Core color	x		x
Type "x" means that role has given type			
Normal or Abstract class		x	x
Interface			
Abstract class			
Interface or Abstract	x		
Elements [(Number of elements) role to share the elements with if there is any]			
Private atribute(s)			
Public method(s)	[(1..*) Proxy], [(1..*) Real S.]	[(1..*) Subject]	[(1..*) Subject]
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)			
Inheritance or Realization	[p. to (1..*) Proxy], , [p. to (1..*) Real S.]	[ch. to (1..*) Subject]	[ch. to (1..*) Subject]
[container to/contained by (Cardinality) Related role]			
Composition			
Dependency			
Aggregation			
Association		[cond. by (1..*) Proxy]	[cor. to (1..*) Real S.]

Table A.7: Facade

Name	Facade	Subsystem
Core role	x	
Type "x" means that role has given type		
Normal class	x	
Non-important		x
Elements [(Number of elements) role to share the elements with if there is any]		
Private atribute(s)		
Public method(s)	[(1..*) Subsystem]	
Relationships [container to/contained by (Cardinality) Related role]		
Composition		
Dependency		
Aggregation		
Association	[cor. to (1..*) Subsystem]	[cond. by (1..*) Facade]

Table A.8: Flyweight

Name	Flyweight	Concrete Flyweight	Flyweight Factory	Client (Optional)
Core role	x		x	
Type "x" means that role has given type				
Normal class				x
Normal or Abstract class		x	x	
Interface	x			
Abstract class				
Interface or Abstract class				
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)		(1..*)		
Public method(s)	[(1..*) Concr. F.]	[(1..*) Flyweight]	(1..*)	
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)	[p. to (1..*) Concr. F.]	[ch. to (1..*) Flyweight]		
Inheritance or Realization				
[container to/contained by (Cardinality) Related role]				
Composition				
Dependency				
Aggregation	[cond. by (1..*) Flyweight F.]		[cor.to (1..*) Flyweight]	
Association		[cond. by (1..*) Client]	[cond. by (1..*) Client]	[cor. to (1..*) Flyweight F.], [cor. to (1..*) Concr. F.]

Table A.9: Class Adapter (inheritance)

Name	Target	Adapter	Adaptee	Client
Core role		x	x	
Type "x" means that role has given type				
Normal class		x		x
Interface	x			
Abstract class				
Normal or Abstract class			x	
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*)Adapter]	[(1..*) Target]	(1..*)	
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)		[ch. to (1..1) Adaptee]	[p.to (1..*) Adapter]	
Realization (implemented)	[p.to (1..*) Adapter]	[ch. to (1..*) Target]		
Inheritance or Realization				
[container to/contained by (Cardinality) Related role]				
Composition or Aggregation				
Dependency				
Aggregation				
Association	[cond. by (1..*) Client]			[cor. to (1..*) Target]

Table A.10: Object Adapter (composition)

Name	Target	Adapter	Adaptee	Client
Core role		x	x	
Type "x" means that role has given type				
Normal class			x	x
Interface				
Normal or Abstract class		x		
Interface or Abstract class	x			
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*)Adapter]	[ch. to (1..*) Target	(1..*)	
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)				
Inheritance or Realization	[p.to (1..*) Adapter]	[ch. to (1..*) Target		
[container to/contained by (Cardinality) Related role]				
Composition or Aggregation		[cor. to (1..*) Adaptee]	[cond. by (1..*) Adapter]	
Dependency				
Aggregation				
Association	[cond. by (1..*) Client]			[cor. to (1..*) Target]

Table A.11: Bridge

Name	Abstraction	Refined Abstraction	Implementor	Concrete Implementor
Core role		x	x	
Type "x" means that role has given type				
Normal class		x		x
Interface				
Abstract class				
Interface or Abstract class	x		x	
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*) Refined A.]	[(1..*) Abstraction]	[(1..*) Concrete I.]	[(1..*) Implementor]
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)				
Inheritance or Realization	[p. to (1..*) Refined A.]	[ch. to (1..*) Abstraction]	[p. to (1..*) Concrete I.]	[ch. to (1..*) Implementor]
[container to/contained by (Cardinality) Related role]				
Composition or Aggregation	[cor. to (1..*) Implementor]		[cond. by (1..*) Abstraction]	
Dependency				
Aggregation				
Association				

Table A.12: Composite

Name	Component	Leaf	Composite	Client
Core role	x		x	
Type "x" means that role has given type				
Normal class				x
Normal or Abstract class		x	x	
Interface	x			
Abstract class				
Interface or Abstract class				
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*) Leaf], [p. to (1..*) Composite]	[(1..*) Component]	[(1..*) Component]	
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)	[p. to (1..*) Leaf], [p. to (1..*) Composite]	[ch. to (1..*) Component]	[ch. to (1..*) Component]	
Inheritance or Realization				
[container to/contained by (Cardinality) Related role]				
Composition or Aggregation	[cond. by (1..*) Composite]		[cor. to (1..*) Component]	
Dependency	[cond. by (1..*) Client]			[cor. to (1..*) Component]
Aggregation				
Association				

Table A.13: Decorator

Name	Component	Concrete Component	Decorator	Concrete Decorator (optional)
Core role	x		x	
Type "x" means that role has given type				
Normal class				x
Normal or Abstract class		x	x	
Interface	x			
Abstract class				
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*) Concrete C.], [(1..*) Decorator]	[(1..*) Component]	[(1..*) Component], [(1..*) Concrete D.]	[(1..*) Decorator]
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)			[p. to (1..*) Concrete D.]	[ch. to (1..1) Decorator]
Realization (implemented)	[p. to (1..*) Concrete C.], [p. to (1..*) Decorator]	[ch. to (1..*) Component]	[ch. to (1..*) Component]	
Inheritance or Realization				
[container to/contained by (Cardinality) Related role]				
Composition	[cond. by (1..*) Decorator]		[cor. to (1..*) Component]	
Dependency				
Aggregation				
Association				

Table A.14: Chain of Responsibility

Name	Handler	Concrete Handler	Client
Core role	x		
Type "x" means that role has given type			
Normal class		x	x
Interface			
Abstract class			
Interface or Abstract	x		
Elements [(Number of elements) role to share the elements with if there is any]			
Private atribute(s)			
Public method(s)	[(1..*) Concrete H.]	[ch. to (1..*) Handler]	
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)			
Inheritance or Realization	[p. to (1..*) Concrete H.]	[ch. to (1..*) Handler]	
[container to/contained by (Cardinality) Related role]			
Dependency			
Aggregation	[cor. and cond. (1..*) Handler]		
Association	[cond. by (1..*) Client]		[cor. to (1..*) Handler]

Table A.15: Command

Name	Command	Concrete Command	Client	Invoker	Receiver
Core role	x				x
Type "x" means that role has given type					
Normal class		x	x	x	x
Interface	x				
Abstract class					
Normal or Abstract class					
Elements [(Number of elements) role to share the elements with if there is any]					
Private atribute(s)		(1..*)			
Public method(s)	[(1..*) Concr. C]	[(1..*) Command]			(1..*)
Relationships [parent/child to (Cardinality) Related role]					
Inheritance (extended)					
Realization (implemented)	[p. to (1..*) Concr. C.]	[ch. to (1..*) Command]			
Inheritance or Realization					
[container to/contained by (Cardinality) Related role]					
Dependency		[cond. by (1..*) Client]	[cor. to (1..*) Concr. C]		
Aggregation	[cond. by (1..*) Invoker]			[cor. to (1..*) Command]	
Association		[cor. to (1..*) Receiver]	[cor. to (1..*) Receiver]		[cond. by (1..*) Concr. C.], [cond. by (1..*) Client]

Table A.16: Interpreter

Name	Abstract Expression	Terminal Expression	Nonterminal Expression	Client	Context
Core role	x				x
Type "x" means that role has given type					
Normal class		x	x	x	x
Interface					
Abstract class					
Interface or Abstract class	x				
Elements [(Number of elements) role to share the elements with if there is any]					
Private atribute(s)					
Public atribute(s)					
Public method(s)	[(1..*) Term. E.], [(1..*) Nonterm. E.]	[(1..*) Abst. E.]	[(1..*) Abst. E.]		
Relationships [parent/child to (Cardinality) Related role]					
Inheritance (extended)					
Realization (implemented)					
Inheritance or Realization	[p. to (1..*) Term. E.], [p. to (1..*) Nonterm. E.]	[ch. to (1..*) Abst. E.]	[ch. to (1..*) Abst. E.]		
[container to/contained by (Cardinality) Related role]					
Dependency					
Aggregation	[cond. by (1..*) Nonterm. E.]		[cor. to (1..*) Abst. E.]		
Association	[cond. by (1..*) Client]			[cor. to (1..*) Abst. E.], [cor. to (1..*) Context]	[cond. by (1..*) Client]

Table A.17: Iterator

Name	Iterator	Concrete Iterator	Aggregate	Concrete Aggregate	Client
Core role	x		x		
Type "x" means that role has given type					
Normal class		x		x	x
Interface	x				
Abstract class					
Interface or Abstract class			x		
Elements [(Number of elements) role to share the elements with if there is any]					
Private atribute(s)					
Public method(s)	[(1..*) Concr. I.]	[(1..*) Iteator]	[(1..*) Concr. A.]	[(1..*) Aggregate]	
Relationships [parent/child to (Cardinality) Related role]					
Inheritance (extended)					
Realization (implemented)	[p .to (1..*) Concr. I.]	[ch. to (1..*) Oterator]			
Inheritance or Realization			[p. to (1..*) Concr. A.]	[ch. to (1..*) Aggregate]	
[container to/contained by (Cardinality) Related role]					
Dependency					
Aggregation					
Association	[cond. by (1..*) Client]	[cor. to (1..*) Concr. A.]	[cond. by (1..*) Client]	[cond. by (1..*) Concr. I.]	[cor. to (1..*) Agregate], [cor. to (1..*) Iterator]

Table A.18: Mediator

Name	Mediator (Optional)	Concrete Mediator	Colleague	Concrete Colleague
Core role		x	x	
Type "x" means that role has given type				
Normal class		x		x
Interface	x			
Abstract class				
Interface or Abstract class			x	
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)				
Public method(s)	[(1..*) Concr. M.]	[(1..*) Mediator]	[(1..*) Concr. C.]	[(1..*) Colleague]
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)	[p. to (1..*) Concr. M.]	[ch. to (1..*) Mediator]		
Inheritance or Realization			[p. to (1..*) Concr. C.]	[ch. to (1..*) Colleague]
[container to/contained by (Cardinality) Related role]				
Dependency				
Aggregation				
Association	[cond. by (1..*) Colleague]	[cor. to (1..*) Concr. C.]	[cor. to (1..*) Mediator]	[cond. by (1..*) Concr. M.]

Table A.19: Memento

Name	Memento	Originator	Caretaker
Core role	x		
Type "x" means that role has given type			
Normal class	x	x	x
Interface			
Abstract class			
Normal or Abstract class			
Elements [(Number of elements) role to share the elements with if there is any]			
Private atribute(s)	(1..*)	(1..*)	
Public method(s)	(1..*)	(1..*)	
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)			
Inheritance or Realization			
[container to/contained by (Cardinality) Related role]			
Dependency	[cond. by (1..*) Originator]	cor. to (1..*) Memento]	
Aggregation	[cond. by (1..1) Caretaker]		[cor. to (1..*) Memento]
Association			

Table A.20: Observer

Name	Subject	Concrete Subject	Observer	Concrete Observer
Core role	x		x	
Type "x" means that role has given type				
Normal class		x		x
Interface				
Abstract class				
Interface or Abstract class	x		x	
Elements [(Number of elements) role to share the elements with if there is any]				
Private atribute(s)		(1..*)	[(1..*) Concr. O.]	[(1..*) Observer]
Public method(s)	(1..*)	(1..*)	[(1..*) Concr. O.]	[(1..*) Observer]
Relationships [parent/child to (Cardinality) Related role]				
Inheritance (extended)				
Realization (implemented)				
Inheritance or Realization	[p. to (1..*) Concr. S.]	[ch. to (1..*) Subject]	[p. to (1..*) Concr. O.]	[ch. to (1..*) Observer]
[container to/contained by (Cardinality) Related role]				
Dependency				
Aggregation				
Association	[cor. to (1..*) Observer]		[cond. by (1..*) Subject]	

Table A.21: State

Name	State	Concrete State	Context
Core role	x		x
Type "x" means that role has given type			
Normal class		x	x
Interface or Abstract class	x		
Abstract class			
Normal or Abstract class			
Elements [(Number of elements) role to share the elements with if there is any]			
Private atribute(s)			
Public method(s)	[(1..*) Concr. S.]	[(1..*) State]	(1..*)
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)			
Inheritance or Realization	[p. to (1..*) Concr. S.]	[ch. to (1..*) State]	
[container to/contained by (Cardinality) Related role]			
Dependency			
Aggregation or Composition	[cond. by (1..*) Context]		[cor. to (1..*) State]
Association			

Table A.22: Strategy

Name	Strategy	Concrete Strategy	Context
Core role	x		x
Type "x" means that role has given type			
Normal class		x	x
Interface	x		
Abstract class			
Inteface or Abstract class			
Elements [(Number of elements) role to share the elements with if there is any]			
Private atribute(s)			(1..*)
Public method(s)	[(1..*) Concr. S.]	[(1..*) Strategy]	(1..*)
Relationships [parent/child to (Cardinality) Related role]			
Inheritance (extended)			
Realization (implemented)	[p. to (1..*) Concr. S.]	[ch. to (1..*) Startegy]	
Inheritance or Realization			
[container to/contained by (Cardinality) Related role]			
Dependency			
Aggregation or Composition	[cond. by (1..*) Context]		[cor. to (1..*) Strategy]
Association			

Table A.23: Template Method

Name	Abstract Class	Concrete Class
Core role	x	
Type "x" means that role has given type		
Normal class		x
Interface		
Abstract Class	x	
Interface or Abstract class		
Elements [(Number of elements) role to share the elements with if there is any]		
Private atribute(s)		
Public method(s)	[(1..*) Concr. C.]	[(1..*) Abstract C.]
Relationships [parent/child to (Cardinality) Related role]		
Inheritance (extended)	[p. to (1..*) Concr. C.]	[ch. to (1..1) Abstract C.]
Realization (implemented)		
Inheritance or Realization		

Table A.24: Visitor

Name	Visitor (Optional)	Concrete Visitor	Element	Concrete Element	Object Structure
Core role		x	x		x
Type "x" means that role has given type					
Normal class		x		x	x
Interface	x				
Abstract class					
Interface or Abstract class			x		
Elements [(Number of elements) role to share the elements with if there is any]					
Private atribute(s)					
Public method(s)	[(1..*) Concr. Visitor]	[(1..*) Visitor]	[(1..*) Concr. Element]	[(1..*) Element]	
Relationships [parent/child to (Cardinality) Related role]					
Inheritance (extended)					
Realization (implemented)	[p. to (1..*) Concr. Visitor]	[ch. to (1..*) Visitor]			
Inheritance or Realization			[p. to (1..*) Concr. Element]	[ch. to (1..*) Element]	
[container to/contained by (Cardinality) Related role]					
Dependency					
Aggregation			[cond. by (1..*) Object Struct.]		[cor. to (1..*) Element]
Association					