QUANTITATIVE ASSESSMENT OF THE FUNCTIONAL EFFECTIVENESS OF DESIGN PATTERNS ON THE PRESENCE OF CODE SMELLS

BY

MAHMOUD ABDULKARIM ALFADEL

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

SOFTWARE ENGINEERING

MAY 2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **MAHMOUD ABDULKARIM ALFADEL** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING**.

Thesis Committee

Dr. Khalid Aljasser (Adviser)

Dr. Mohammad Alshayeb (Member)

Dr. Sajjad Mahmoud (Member)

Dr. Khalid Aljasser
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

Date

©Mahmoud Abdulkarim Alfadel
2017

I dedicate my thesis work to my family and friends. A special feeling of gratitude to my loving parents, whose words of encouragement and push for tenacity ring in my ears. An exclusive dedication to my brothers, sisters, and friends. This work would not be possible without their support. I would also like to sincerely thank all teachers who taught me, as their encouragements and motivations have laid the foundation for this work.

ACKNOWLEDGMENTS

First of all, I would like to thank almighty Allah for giving me the ability and strength to work on and accomplish this thesis. I wish to give my sincere appreciation to my advisor Dr. Khalid Aljasser for all his efforts and countless hours of reflecting, reading, encouraging, guiding, and most of all patience throughout the entire work of this thesis. He has been my mentor, teacher and father. I feel very lucky to have him as supervisor for my thesis.

I am also grateful for the committee members of my thesis: Dr. Mohammad Alshayeb and Dr. Sajjad Mahmoud who were more than generous with their expertise and precious time. They have been dedicating valuable time out of their busy schedule in order to provide useful feedback to improve the quality of this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENT **iii**

LIST OF TABLES **vii**

LIST OF FIGURES **viii**

ABSTRACT (ENGLISH) **ix**

ABSTRACT (ARABIC) **xii**

CHAPTER 1 INTRODUCTION **1**

1.1 Overview 1

1.2 Research Problem 3

1.3 Research Motivation 3

1.4 Problem Statement 5

1.5 Research Methodology 6

1.6 Thesis Objectives 7

1.7 Thesis Contributions 8

1.8 Thesis Outline 9

CHAPTER 2 BACKGROUND **10**

2.1 Design Patterns 10

2.2 Code Smells 12

2.3 Design Pattern Detection Tools 14

2.3.1 DeMIMIA 17

2.3.2 DP-Miner . . . 18
2.3.3 DPRE . . . 18
2.3.4 FUJABA . . . 19
2.3.5 MARRPLE . . . 19
2.3.6 Pinot . . . 20
2.3.7 PTIDEJ . . . 20
2.3.8 Tsantalis . . . 21
2.3.9 SPQR . . . 21
2.3.10 Web of Pattern(WOP) . . . 22
2.3.11 DPJF . . . 22
2.3.12 Summary . . . 23
2.4 Code Smell Detection Approaches . . . 24
2.4.1 Manual Approach . . . 25
2.4.2 Metrics-based Approach . . . 25
2.4.3 Search-based Approach . . . 26
2.4.4 Symptoms-based Approach . . . 26
2.4.5 Visualization-based Approach . . . 27
2.4.6 Probabilistic-based Approach . . . 27
2.4.7 Cooperative-based Approach . . . 27

CHAPTER 3 LITERATURE REVIEW **29**
3.1 Design Patterns Impact on Software Quality Attributes . . . 30
3.2 Code Smells Impact on Software Quality Attributes . . . 33
3.3 The Relationship between Design Patterns and Code Smells . . . 35
3.3.1 Structural Relationship . . . 36
3.3.2 Refactoring Relationship . . . 37
3.3.3 Co-occurrence Relationship . . . 39
3.4 Summary . . . 40

CHAPTER 4 EXPIREMENTAL SETUP **42**
4.1 Data Collection . . . 42

4.1.1 Subject Systems 42
4.1.2 Data of Patterns and Smells 44
4.2 Methodology Description 48
4.2.1 Odd Ratio (OR) Test 51
4.2.2 Wilcoxon Signed-Rank Test 51
4.2.3 Mann-Whitney U Test 52
4.2.4 Kruskal-Wallis Test 54

CHAPTER 5 EXPERIMENTAL RESULTS 55

5.1 Functional Effectiveness Evaluation of Design Patterns on Code Smells 57
5.1.1 Design Level(Class Level) 57
5.1.2 Category Level 63
5.1.3 Individual Design Patterns Level 67
5.2 Discussion and Analysis 74
5.2.1 Co-occurrence between Design Patterns and Code Smells . 74
5.2.2 Comparison to Related Work 77
5.2.3 Threats to Validity 80

CHAPTER 6 CONCLUSION AND FUTURE WORK 83

6.1 Conclusion 83
6.2 Future Work 86

REFERENCES 88

VITAE 108

LIST OF TABLES

2.1 Java code smells proposed by Fowler [1] 15
2.2 Design Pattern detection tools 24
2.3 Code Smells detection approaches 28

4.1 Projects statistics 45
4.2 Patterns instances in the subject systems 46
4.3 Events involved in OR test 51
4.4 Test of normality 53
4.5 Test of homogeneity of variance 54

5.1 Descriptive statistics of smells in each system 56
5.2 Odd Ratio test analysis for smell-proneness evaluation of participant vs. non-participant design pattern classes groups 58
5.3 Statistics of smell proneness in all systems 61
5.4 Statistics of smell density in all systems 62
5.5 P-values of evaluation design pattern categories using Kruskal-Wallis test 65
5.6 P-values of evaluation design pattern categories using Mann Whitney test 67
5.7 P-values obtained from Kruskal Wallis test of each design pattern category 70
5.8 Data of individual patterns with individual smells 71
5.9 Extraction of filtered association rules 74

LIST OF FIGURES

2.1 Representation of Command Design Pattern in WebMail system [2] 12
2.2 Relationships between detected and existing smells in a system . . 17

3.1 The common attributes influenced by Design Patterns 30
3.2 The common attributes influenced by Code Smells 35
3.3 The common relations between Design Patterns and Code Smells 40

4.1 Participant to non-participant classes 45
4.2 Methodology flow 48

5.1 Smell proneness comparison of participant vs. non-participant design pattern classes in all systems 60
5.2 The respective values of smell proneness in SDP and SnDP . . . 61
5.3 The respective values of smell density in SDP and SnDP 63
5.4 Smell proneness comparison of the design patterns categories . . . 66
5.5 Comparison of smell proneness in creational category 68
5.6 Comparison of smell proneness in structural category 69
5.7 Comparison of smell proneness in behavioural category 69

THESIS ABSTRACT

NAME:	Mahmoud Abdulkarim Alfadel
TITLE OF STUDY:	QUANTITATIVE ASSESSMENT OF THE FUNCTIONAL EFFECTIVENESS OF DESIGN PATTERNS ON THE PRESENCE OF CODE SMELLS
MAJOR FIELD:	Software Engineering
DATE OF DEGREE:	May 2017

Software systems are often developed in a way that good practices of the object-oriented paradigm are not fulfilled, causing the occurrence of specific dis-harmonies which called code smells. Code bad smells are indicators of poor solutions in a fragment of code that propose a potential problem needs to pay attention in code or design. On the other hand, design patterns are intended to catalogue the best practices for developing object-oriented software systems. Although apparently widely divergent, there may be an co-occurrence relation between design patterns and code smells, since this phenomenon is sometimes mentioned in studies in the field of software engineering, albeit discreetly. Hence, this work carries out an investigative study and analysis with the intention of identifying the relationship

between design patterns and code smells in systems, that may happen due to the inadequate use of design patterns. The work is aimed to have the following purposes: (1) to evaluate empirically if the presence of design patterns connects to the presence of code smells in the class level. (2) to evaluate empirically if the relation between each category of design pattern and code smells would hold as in the class level. (3) to evaluate empirically the relation between individual design patterns and code smells. To accomplish this work, we first perform a literature review in order to understand the recent states concerning design patterns and code smells, then, we accomplish an empirical study using twenty design patterns and thirteen code smells in ten small-size to medium-size, open source Java systems. Specifically, we evaluate statistically the presence of design patterns and their possible usage effects on code smells that lead to their co-occurrences. The study is conducted through three levels: (1) Class level, (2) Design Pattern Category level. (3) Pattern level. Association rules are applied to extract the strong rules that describe the relation between pattern-smell pairs. The observed relationship concludes that classes participating in design patterns display less smell proneness and smell density than classes not participating in design patterns in the majority of systems. It was discovered that every one of the different categories acts in the same way in terms of smell proneness. This observation tends towards the conclusion that the adoption of any of the design patterns might produce the same reliable software. By examining the presence of individual smells in code, it is possible to discover the most common ones: Blob, God Class and

External Duplication. The majority of the important rules we discovered combine the presence of a design pattern, and the absence of a code smell. There are rules, however, which represent patterns that are linked with certain smells. Singleton, State, Strategy, Adapter and Decorator are some of the patterns that are generally not collocated with smells, while for Command and Memento this association is significantly weaker. Command patterns are linked to the Blob, External Duplication and God Class smells, whereas the Memento patterns are connected with Blob and External Duplication, with the God Class being the exception. In fact, design patterns misuse could potentially facilitate the production of bad smells. The most noteworthy cases were the co-occurrences of the Command design pattern with the Blob and the God Class. Although professional developing of software systems is cataloged in design patterns, this observation could be utilized to pay developers' attention for the possible undesired sound effects when design patterns are applied inadequately.

ملخص الرسالة

الاسم الكامل: محمود عبدالكريم الفاضل

عنوان الرسالة: التقييم الكمي للفعالية الوظيفية لأنماط التصميم وتأثيرها على وجود عيوب الرماز

التخصص:علوم الحاسب الآلي

تاريخ الدرجة العلمية: مايو 2017

إن تطوير أنظمة البرمجيات يتم غالبا بطريقة غير ملائمة للممارسات الجيدة في البرمجة غرضية التوجه ، مما قد يتسبب في حدوث ما يسمى بعيوب الرماز. عيوب الرماز هي مؤشرات ضعف الحلول في جزء من التعليمات البرمجية التي تقترح مشكلة محتملة تحتاج إلى إيلاء الاهتمام في التعليمات البرمجية أو التصميم. من ناحية أخرى، فإن أنماط التصميم تهدف إلى الفهرسة الأفضل لتطوير أنظمة برمجيات غرضية التوجه. قد تكون هناك علاقة مشتركة بين أنماط التصميم وعيوب الرماز، حيث أن هذه الظاهرة مذكورة في بعض الأحيان في الدراسات الخاصة في مجال هندسة البرمجيات. وبالتالي، فإن هذا العمل يقوم بعملية التحليل والتقييم بقصد تحديد العلاقة بين أنماط التصميم وعيوب الرماز في أنظمة البرمجيات، التي قد تحدث بسبب الاستخدام الخاطئ لأنماط التصميم. ويهدف العمل إلى ما يلي: (1) التقييم التجريبي ما إذا كان وجود أنماط التصميم قد يتصل بوجود عيوب الرماز في مستوى الصف. (2) التقييم التجريبي ما إذا كانت العلاقة بين كل فئة من فئات أنماط التصميم وعيوب الرماز مشابهة لما هي عليه في مستوى الصف. (3) التقييم التجريبي للعلاقة بين كل نمط تصميم على حدة وعيوب الرماز . لإنجاز هذا العمل،قمنا بمسح للدراسات المرجعية ذات الصلة بأنماط التصميم وعيوب الرماز, لاحقاً, قمنا ببناء دراسة تجريبية على 10 أنظمة برمجية مفتوحة المصدر بلغة Java تتراوح مابين صغيرة إلى متوسطة الحجم .على وجه التحديد، قمنا إحصائياً بتقييم علاقة الوجود المشترك بين أنماط التصميم وعيوب الرماز. تم إنجاز العمل من خلال ثلاث مراحل: (1) مرحلة تصميم الصفوف. (2) مرحلة فئات أنماط التصميم. (3) مرحلة كل نمط من أنماط التصميم على حدة. تم تطبيق قواعد الترابط لاستخراج العلاقات القوية بين أنماط التصميم وعيوب الرماز. تخلُص النتائج إلى أن الصفوف المشاركة في أنماط التصميم تعرض عيوباً في الرماز أقل من تلك التي تعرضها الصفوف الغير مشاركة

في انماط التصميم وذلك في غالبية الانظمة البرمجية المستخدمة في هذه الدراسة. بالإضافة لذلك, يجدر الإشارة إلى أن أنماط التصميم لها نفس التأثير على عيوب الرماز. إن هذه النتيجة قد تدل على أن اعتماد أي فئة من فئات أنماط التصميم قد ينتج نفس البرنامج من ناحية جودة الوثوقية. من خلال دراسة عيوب الرماز يمكن اكتشاف العيوب الأكثر شيوعا في هذه الدراسة: Blob, God Class, External Duplication. إن غالبية قواعد الترابط التي تم اكتشافها تشير إلى أن وجود أنماط التصميم في الصفوف يقود إلى عدم وجود عيوب في الرماز. من جهة أخرى, فإن هناك بعض قواعد الترابط التي أظهرت عكس النتيجة السابقة حيث أن بعض أنماط التصميم لها علاقة وثيقة ببعض عيوب الرماز. أنمام التصميم: Singleton, State, Strategy, Adapter and Decorator أظهرت بشكل فعّال أنه لا يوجد ترابط بينها وبين عيوب الرماز فيما أظهرت أنماط التصميم: Command, Memento العلاقة الوثيقة بينها وبين بعض عيوب الرماز وبالأخص: Blob, God Class, External Duplication. في الواقع فإن الاستخدام الخاطئ لأنماط التصميم قد يفضي إلى حدوث عيوب في الرماز. من الجدير بالذكر قوة العلاقة بين نمط التصميم Command وعيبي الرماز Blob, God Class. على الرغم من المهنية الاحترافية التي يُشار إإليها عند استخدام أنماط التصميم, فإن النتائج الملاحظة في هذه الدراسة يمكن أن تُستخدم كحافز لدفع اهتمام المطورين البرمجيين باتجاه التأثيرات الغير مرغوب بها والناتجة عن تطبيق أنماط التصميم بشكل غير مناسب.

CHAPTER 1

INTRODUCTION

1.1 Overview

Design patterns (DPs) are general reusable structures and features to recurring problems in a software design [3]. They aim to increase flexibility, maintainability and reusability. Thus, a design pattern is a guide of solving problems in systems. Design Pattern usage has a great influence on the software quality attributes i.e. maintainability including changeability and understandability, reusability and fault proneness. They have been cataloged for reusability purposes [3]. DPs are classified into three categories, structural patterns, behavioral patterns and creational patterns. The GoF book [3] of design patterns is a good resource of object oriented design practices. There are 23 DPs scattered among the three categories.

Since their inception, design patterns have been employed by designers and developers who realized how patterns could improve development. There is broad proof that design patterns may affect software quality attributes positively and

negatively [4, 5, 6, 7]. Therefore, design patterns are typically beneficial, provided that they are properly implemented. On the other hand, as design patterns became trendy, they are used by software developers as an exploratory means. The inappropriate employment of design patterns can lead software system code to flaws, i.e. bad smells. Bad smells are signs of a lack design in a part of code that presents a problem in a system design or code [1]. Fowler's book [1] sets the foundation stone of the code smells where 22 code smells have been identified. Fowler has defined the code smells as code blocks need refactoring. Kerievsky [8] and Marinescu et al. [9] expanded a set of bad smells. Kerievsky [8] called attention to the use of DPs as a refactoring method to resolve code smells which are called disharmonies. Marinescu et al. [9] provided a guide of bad smells referred as "disharmonies". Fouste et al. [10] stated that code smells are better tool for developer than metrics, since they depend on a specific programming style. On the other hand, metrics are used to detect over code smells. In the context of our study, the essential objective is to explore if there is a link between design patterns and code smells. However, based on the conducted literature, studies have rarely addressed the relationships between design patterns and code smells sufficiently. In this line, we aim to analyze these cases in order to aid designers and developers to better use design patterns. Our ultimate goal is to discover the possible connections between design patterns and code smells.

1.2 Research Problem

Software design is becoming an important factor which combines classes and patterns. Design patterns play an important role in terms of software design quality. Producing good quality design concerns software designers. On the other hand, design flaws may hamper the aimed purpose. Some flaws known as bad code smells identified by Fowler [1] are poor solutions to recurring design problems. According to the literature, the characteristics of design patterns structure i.e. complexity may have relationships with code smells. The study of the relationships between design patterns and code smells can provide deeper insights into the impact on the quality of the design. However, the co-occurrence of design pattern and code smell has not been fully analyzed and need to be investigated further due to the following reasons:

- The functional effectiveness of design patterns on code smells has a conjecture in the state of the art, and so needs to be fulfilled.
- Only high level (class level) is evaluated, which lead to a need of evaluating the other levels i.e category level and pattern level.

1.3 Research Motivation

At first glance, while design patterns are associated with good software design and code, bad smells define lack of design or any code flaw. When design patterns are not used in a proper, they may degrade the system performance and also turn code

more complex unnecessarily manner [5, 11, 12, 13]. This work aims at identifying design patterns inappropriate employments and their consequences. In order to focus on a specific scope, this work exploits the occurrence of bad smells in code that is part of a design pattern.

Design patterns and code smells are topics of periodic research. Despite this frequency, they are rarely investigated in the same research context since they represent antagonistic structures. Therefore, the first step is to understand how studies relate these topics. Among these studies, some tools [9, 14, 15, 16] have been proposed to spot code parts for refactoring to DPs. Some other studies [17, 18] propose the relationship between DPs and code smells in terms of structural relationship.

Moreover, some conjectures in the literature recommend that using design patterns is not always the preferred choice. In addition, the improper employment of a DP can yet initiate code smells [4, 11, 19]. For instance, McNatt et al. [19] analyzed the advantages and disadvantages that affect quality attributes like maintainability, factorability, and reusability. Design patterns have their place, but their inadequate use may increase complexity and decrease some quality attributes. The main purpose of this thesis is to empirically evaluate the relationship between design patterns and code smells in software engineering systems.

1.4 Problem Statement

In order to cover the objectives of this work, functional effectiveness of design patterns on code smells is evaluated in different granularity levels, as follows:

- **Design Level**

  This level empirically evaluates the differences of functional effectiveness between classes involved in any design pattern and classes that are not involved in any design patterns in terms of smell proneness and smell density i.e. (Smelly Design Pattern(SDP) versus Smelly non-Design Pattern(SnDP)).

- **Category Level**

  This level empirically evaluates the differences of functional effectiveness among classes that involve in different categories of design patterns in terms of smell proneness.

- **Pattern Level**

  This level empirically evaluates the functional effectiveness of classes that involve in a specific single design pattern individually, in terms of smells.

According to the previous levels, we formulate the research questions to be answered as the following:

RQ1: Are design pattern classes more smell-prone than the non-design pattern classes in software systems?

RQ2: Do code smells have significant differences when they present in the different categories of design pattern classes?

RQ3: Are the participant classes in a specific individual design pattern more smell-prone in specific smells than other ones?

1.5 Research Methodology

This section describes the research approach as follows:

Phase 1: Comprehensive Literature Review

A comprehensive literature review is conducted to survey the existence empirical evidence that addressed the quality of design patterns specially design pattern modularity, smell proneness and smell density.

Phase 2: Identifying popular design patterns detection tools

This phase identifies a tool or more that help in detecting instances of design patterns. To do so, the available popular tools are identified. Then these tools are evaluated. After that, one or more of these tools are chosen to be used in this research.

Phase 3: Identifying popular code smells detection tools

The objective of this phase is to identify a tool or more that help in detecting instances of code smells. To do so, the available popular tools are identified. Then these tools are evaluated. After that, one or more of these tools are chosen to be used in this research.

Phase 4: Data collection and experimental setup

After conducting a survey for the literature of the existing empirical evidence and

tools, the data needed for conducting this study is collected and prepared. A group of Java open source systems are prepared for this study. For each class in these systems, the code smells and design patterns data are collected and prepared.

Phase 5: Empirical evaluation of the functional effectiveness of design patterns on code smell proneness in object oriented systems

This work studies the functional properties i.e. modularity of design patterns to be evaluated in terms of code smells. First, the classes in the subject systems are separated into two clusters: a cluster of participant classes and another one of non-participant classes in the design patterns. Then for each group, the functional effectiveness of classes is evaluated and compared in terms of presence of code smells. After that, the effectiveness properties of each category of the design patterns is evaluated and compared.

1.6 Thesis Objectives

The objective of conducting this research work is evaluating the functional effectiveness of design patterns in object oriented systems on the presence of code smells. Some sub-objectives can be extracted as follows:

1. Empirical evaluation of code smell-proneness and smell density in design pattern classes versus non-design pattern classes.

2. Empirical evaluation of code smell-proneness in design pattern classes versus

non-design pattern classes in each category of the design patterns(creational, structural and behavioral).

3. Empirical evaluation of code smell-proneness for the individual design patterns.

1.7 Thesis Contributions

As an expected result of the work presented in this thesis, the following contributions can be highlighted:

- A literature review detaching how studies relate design patterns and bad smells, revealing the lack of studies focusing on the co-occurrence between design patterns and bad smells. This provides the academic community with the state of the art in this field.
- Empirical evaluation of the relationship between design patterns and code smells in the class design level.
- Empirical evaluation of the differences among design pattern categories in terms of smell proneness.
- Empirical evaluation of the differences among individual design patterns in terms of smell proneness.
- Apply data mining techniques to identify real instances of design pattern and bad smell co-occurrences.

- Submitting a paper to a high quality ISI journal.

1.8 Thesis Outline

The reminder of this work is structured as follows. Chapter 2 presents the technical background. The literature review is presented in Chapter 3. Chapter 4 describes the experiment setup of the work. Chapter 5 highlights the experimental results and discussion of the results. Finally, Chapter 6 concludes the thesis, lists the limitations of the work proposed, and recommends a set of directions in which future work can be conducted.

CHAPTER 2

BACKGROUND

The correct understanding of a research work begins with the understanding of the concepts related to it. This chapter presents relevant concepts for this research work. Section 2.1 presents some examples of design patterns where as Section 2.2 introduces and presents some examples of code smells.

2.1 Design Patterns

In 1994, Gamma at al. [3] have written the book of design patterns: "Design Patterns-Elements of Reusable Object Oriented Software" (also known as Gang of Four, or GOF) [3]. 23 DPs are spread among three categories: Creational, Structural and Behavioural. Design patterns are considered as recurring solutions to a design problem in a particular scenario i.e. improve reusability and reduce coupling [5]. Moreover, DPs can ease communication among team members by using terminology instead of using traditional explanatory [6]. They became popular for using in software design. However, an excessive use of DP in a wrong manner may

cause a design problem and hence, introduce a problem in code. There exist three categories of design patterns, Creational, Structural, and Behavioural. Creational design patterns are design patterns that deal with creating objects so that the created objects serve some purpose that suitable to the situation like controlling the object creation. Builder, Factory Method and Prototype are some examples of Creational design pattern. Structural design patterns are design techniques that facilitate software design by identifying simple ways to realize relationships among the different entities. Adapter, Bridge Composite are considered as Structural design patterns. Behavioral design patterns are communication patterns. These patterns ease and increase the flexibility of communication. Command, Iterator and Memento belong to the Behavioural category. In the context of this work, Command design pattern is explained in details. The study by Fehmi et al. [20] observed that Command design pattern plays a correcting role of code smells i.e. SwissArmyKnife. As an example, we will be explaining Command design pattern which is used in several studies.

Command pattern is a behavioral design pattern driven by data. Its intent is to wrap a request as a command and pass it to an invoker object. Command design pattern consists of several classes where each class is playing different role as follows: (1) Command, (2) Concrete-Command, (3) Client, (4) Invoker, and (5) Receiver. A real example of Command DP is included in ***WebMail*** system [2]. Figure 2.1 illustrates part of Command class diagram in WebMail system.

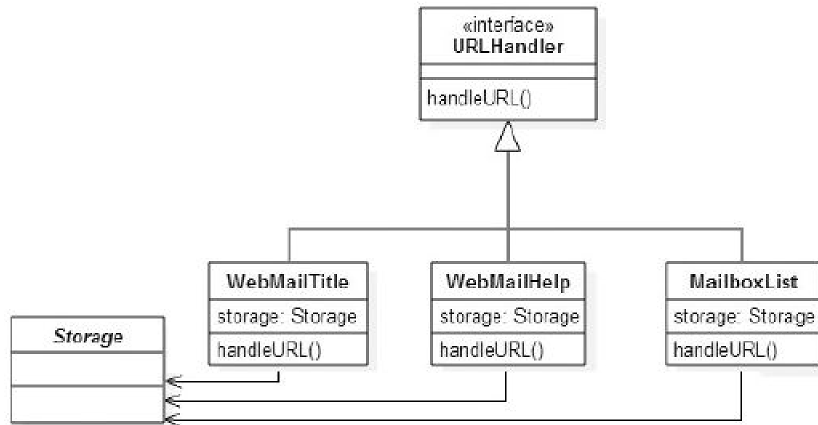


Figure 2.1: Representation of Command Design Pattern in WebMail system [2]

As shown in Figure 2.1, the ***URLHandler*** is the interface which represents the Command i.e. abstract command. On the other hand, three concrete classes as commands implement that interface using ***handleURL( )*** method. These concrete commands instantiate an object of ***Storage*** class. Storage class plays as the classic receiver class in Command DP. As shown in Figure 2.1, the method ***handleURL( )*** is the corresponding method to the classic method called "execute" in the traditional definition of Command DP.

2.2 Code Smells

Code smells can be referred as problems that appear in a fragment of code that make software hard to maintain and change [1]. Several classifications have been associated with bad smells such as code smells and design smells [21]. Anti-patterns are other concepts which can be linked to code smells. Fehmi et al. [20] stated that code smells are more related to the inner scope of classes while anti-patterns are related to the relationships included among classes. Webster [22]

has written the first book on anti-patterns in 1995. 40 anti-patterns have been described by Brown et al. [23]. According to Brown et al., anti-patterns represent bad designs when they are employed in massive manners. So, in this work, we consider code smells and anti-patterns as synonyms. For example, Blob anti-pattern which was introduced by Brown et al. [23] is similar to Large Class smell introduced by Fowler et al. [1]. To get more understanding of code smells, we introduce a small story which is experienced by Liliana [24]. Liliana is an expert programmer that participated in developing Tomcat project[1]. When Liliana tried to add functionality to JNDIrealm class in the project, she reported a problem that belongs to that class. The class JNDIrealm has multiple methods that look like in Listing 2.1.

Listing 2.1: Java code includes Data Clumps smell

```java
boolean compareCredential(DirContext context, String credentials)
   {
        /*sync since super.digest() does this same thing*/
        synchronized(this)
        password = password.substring(5);
        md.reset();
        String digestedPassword = new String((md.digest()));
        validated = password.equals(digestedPassword);
   }
```

The code in Listing 2.1 has the parameters: context and credentials. In fact, Liliana discovered during her inspection of the code that these parameters appear together in the parameters list of other seven methods. Consequently, she concluded that an encapsulation does fit in this case. According to her, this was

[1]http://tomcat.apache.org/

beneficial because productivity will improve accordingly. The case where Liliana noticed a problem is a smell called: **Data Clumps**. Data Clumps is a smell where it appears when a group of objects are used together in different places through out code. This smell might need an attention since it makes software difficult to maintain because if any changes of an object from the group are needed, there should be a need to examine every location where the group of objects appear to check if it needs a change.

Another important code smell is Duplicated Code. According to Zhang et al. [25], Duplicated Code smell can occur very common in systems since Fowler et al. [1] identified code smells (2000 to 2009 inclusive). This fact supports the idea of Fowler et al. [1] that "Number one in the stink parade is duplicated code". Code Duplication can also be in two or more totally unrelated classes or even in unrelated software systems as stated in Oliveira et al [26]. This situation is the hardest to be detected and definitely the most difficult to be refactored. Most of smells which identified in the literature have been proposed by Fowler [1]. More smells have been subsequently proposed by Emen at al. [27]. A list of some interesting smells is shown in Table 2.1

2.3 Design Pattern Detection Tools

Software quality attributes are expensive activities in software engineering. For instance, in order to achieve the maintainability tasks, reverse engineering can play an important role. Reverse engineering provides an abstract view from a

Table 2.1: Java code smells proposed by Fowler [1]

Smell Name	Smell Description
Duplicated Code	Identical code appears in more than one place.
Feature Envy	When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air.
Message Chain	Occur when a sequence of message calls or temporary variables present to get a desired object.
Switch Statement	When the same **switch statement** or ("if...else if...else if") is duplicated across a system.
Long Method	When a method is too long which makes it difficult to understand and so maintain.
Large/God Class	A class contains too many instance variables and so too many responsibilities.
Primitive Obsession	A program uses primitives like **int** and **strings** instead of domain-specific objects which simplify code.
Comments	Heavily commented code indicates often bad codes. Refactor the code so the comments become well unnecessary.
Data Clumps	Bunches of data that hang around together and might need to be made into their own object.

subject system. Also, design patterns detection is a task which is obtained by reverse engineering in order to identify the patterns that have been applied in a system. For this purpose, many tools have been proposed in the literature. There are many characteristics that should be available in the tool or in the combination of tools that are going to be used to collect design patterns data information for our work. These characteristics are as follows:

- Be able to work on Java source code since that the subject systems are in Java.

- At least half of the patterns in each category should be covered. This is because that one of our objectives of conducting this study is to compare the difference in smell proneness among the different categories of software design patterns.

- A high level of detection accuracy is required (at least 90%). This is because a low level of detection accuracy will negatively affect the planned experiments. There are two essential metrics to evaluate the tool accuracy, **precision** and **recall**. They are calculated as follows:

$$Precision = \frac{|\{\text{existing code smells}\} \bigcap \{\text{detected code smells}\}|}{|\{\text{detected code smells}\}|} \quad (2.1)$$

$$Recall = \frac{|\{\text{existing code smells}\} \bigcap \{\text{detected code smells}\}|}{|\{\text{existing code smells}\}|} \quad (2.2)$$

  Figure 2.2 depicts the relation between detected and existing code smells in a system.

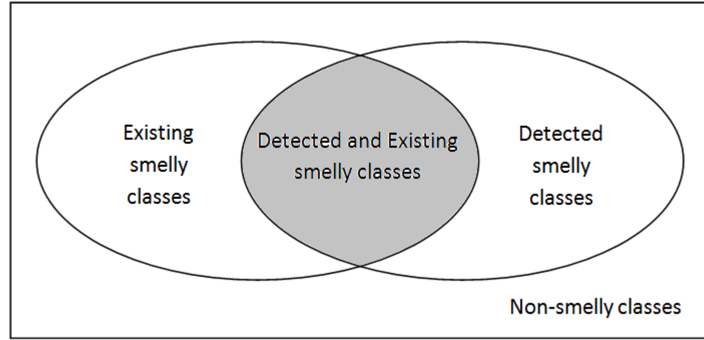


Figure 2.2: Relationships between detected and existing smells in a system

There are many DP detection tools in the literature. The following sections present 11 tools that work on Java language source code. These tools are as follows:

2.3.1 DeMIMIA

DeMIMIA [28] is a semi-automatic tool for identifying pattern-like micro-architecture in Java source code. This tool ensures the traceability of the pattern like micro-architecture between design and implementation. There are three layers involved in the process of DeMIMA. The first and the second layer are to recover an abstract model from the source code. The third layer is to detect patterns from the recovered abstract model [28]. Among DeMIMIA tool advantages, it can provide information related to dependencies of classes in a system.

In evaluating the suitability of this tool for our work, it was found that this tool is not suitable. It can be seen in Table 2.2 (page 22) that the precision of this tool is too low i.e. 34%. This is expected to affect the results of our work. So,

this tool will not be considered.

2.3.2 DP-Miner

DP-Miner [29] is another tool that aim to detect design patterns by defining the structural characteristics of each design pattern. These characteristics are represented in terms of weights and in a matrix. The discovery process includes three major processes: structural analysis, behavioral analysis and semantic analysis. In evaluating the suitability of this tool for our work, it was found that this tool has a good precision and recall. However, as it can be seen in Table 2.2, this tool is designed to detect a very few number of design patterns: Adapter/Command, Bridge, Composite and State/Strategy. At the same time, this tool does not differentiate between the instances of Adapter and Command and between the instances of State and Strategy. So, this tool is not considered in our work.

2.3.3 DPRE

DPRE tool [30] is a two-phase design pattern detection tool. In the first phase, a coarse grained level recovery process is applied to the source code. In this level, the structure of the design patterns is considered and a parsing technique for visual language is utilized as well. In the second phase, a fine grain validation is applied to the retrieved instances in the first phase [30]. The tool has reported a good precision percentages. It reaches %97 in some cases. However, there are two problems in this tool. First, this tool detects structural design patterns only. This

violates the second characteristic mentioned in the beginning of Section 2.3. The second problem (as shown in Table 2.2) is that the precision rate of this tool varies from 62% to 97% and the recall rate is not mentioned. The achieved precision rate is not enough to obtain accurate results and the recall rate is not mentioned to make sure that this tool retrieves only true pattern instances.

2.3.4 FUJABA

FUJABA tool [31] utilizes fuzzy logic and abstract syntax graph to recover design patterns. Different graphs are used to model different design variants models. The fuzzy logic utilizes to cope with the implementation variants by identifying fuzzy rules. They are defined together to handle the implementation variants by giving a degree of belief to test whether a pattern is found or not [31]. On one hand, the tool detects all GoF's patterns in [3] which is a very good advantage. On the other hand, this tool is not evaluated with any performance measures as it can be seen in Table 2.2.

2.3.5 MARRPLE

MARRPLE tool [32] is a design patterns detection tool that consists of four different components. The first component is the information detector engine which collects the required information for pattern detection. The second component is the jointer which extracts the potential design patterns. The third component is the classifier which validates the retrieved design patterns. The last component

is the output generator which is responsible for providing the user with the final output [32].

This tool can detect three patterns only: Abstract Factory, Composite and Visitor. This is not suitable for this work as mentioned in the second characteristic of the required tool in the beginning of Section 2.3. Also, it can be seen in Table 2.2 that the performance measures of this tool is not satisfying enough.

2.3.6 Pinot

Pinot [33] is a tool that reclassify design patterns into different categories claiming that this reclassification facilitates design pattern detection. They use light-weight static analysis for analyzing programs. The tool is not used in this study. The reason for that is the absence of precision and recall rates or any other performance measure of this tool as it can be seen in Table 2.2. So, this tool is not suitable for this work.

2.3.7 PTIDEJ

The PTIDEJ tool [34] is a design pattern detection tool which is developed by PTIDEJ group in Canada. This tool uses constraints solving with explanation to detect design patterns. The tool starts with detecting design patterns that exactly match the predefined instances. Then the constraints are relaxed to allow for detecting more patterns.

As it can be seen in Table 2.2, the tool has very high recall. However, the precision

rates are low. The group of PTIDEJ has validated some design patterns data which detected by the tool.

2.3.8 Tsantalis

Tsantalis is a design pattern detection tool that use similarity scoring between graph vertices to detect design patterns. It also exploits the fact that a design pattern resides in one or more hierarchy [35]. The tools is used commonly in many studies of the literature [36]. It has reported very high precision and recall, 100% and 95%, respectively. However, There are two reasons which lead us to avoid using the tool in our work. First, although this tool can detect 13 patterns, we can only benefit from 9 patterns as it can be seen in Table 2.2. This is because this tool cannot differentiate between the instances of Adapter and Command patterns and between the instances of State and Strategy patterns. The second reason is that this tool does not extract all patterns information. Also, this tool cannot extract all the roles in each design pattern.

2.3.9 SPQR

Smith et al. [37] presented a design pattern detection tool called SPQR. The developers of this tool use formal denotational semantics to encode design pattern elements and to encode the rules by which these elements are combined to form design patterns. Also, these semantics are used to encode the structural and behavioral relationships among the elements of design patterns. In evaluating the

suitability of this tool for our work, it was found that this tool is not suitable. First of all, this tool can only detect one pattern which is the Decorator as it can be seen in Table 2.2. Secondly, this tool does not provide any verification for its performance metrics.

2.3.10 Web of Pattern(WOP)

WOP tool is presented by Dietrich et al. [38] to detect design patterns in software systems. In this tool, an ontology language is used to define design patterns and to create a web of patterns. Then an algorithm for finding exact matches is applied to extract the instances of design patterns. The constraints can also be relaxed to retrieve more potential instances [38]. This tool detects 4 patterns: Abstract Factory, Bridge, Strategy and Adapter. Moreover, the precision and recall rates, 57.3% and 54.5% respectively, are not satisfying as it can be seen in Table 2.2. Hence, the tool is not considered in this work.

2.3.11 DPJF

DPJF is a tool for detecting occurrences of design patterns in Java programs [39]. The acronym DPJF stands for Detection of Patterns by Joining Forces, reflecting that the unusually good results of DPJF are achieved by a novel combination of various well-known static analysis techniques (forces). Nevertheless, we can only benefit from 8 patterns as seen in Table 2.2. Also, this tool does not extract all patterns information i.e this tool cannot extract all the roles in each design

pattern. Interestingly, DPJF [39] and Tsantalis [35] tools have reported very high rates. For precision, both tools reported 100%. On the other hand, they presented about 80% and 95% respectively in terms of the recall rates.

2.3.12 Summary

In the context of our work, choosing a suitable design pattern detection tool is very critical task. Therefore, the literature was surveyed searching for software design pattern detection tools. Section 2.3 provides information related to some DP detection tools. As a result, 11 tools that work on Java language source code are identified in terms of the detectable design patterns along with the precision and recall rates as shown in Table 2.2. We found that design pattern detection tools are not suitable for our work since that all of them are not congruent with the characteristics required to obtain reliable data. These characteristics are mentioned in Section 2.3. The surveyed tools cover a small set of design patterns except for FUJABA [31]. The tools shown in Table 2.2 displays generally low percentages of the accuracy metrics except for Tsnatalis [35] and DPJF [39]. Moreover, none of the DP detection tools provide all the roles of design patterns. Therefore, **P-Mart repository** came across. Several works [40, 41, 42, 43] have used it due to its reliable source and validation. Moreover, it is considered as a benchmark data set in the literature [43]. P-Mart repository was collected based on GoF's book [3]. It consists of 10 systems. Each system patterns collected in a separate session by B.Sc. and M.Sc. students. Also, the collections of other people who

are working in this area are used as well. P-Mart repository is considered as a powerful replacement for design pattern detection tools.

Table 2.2: Design Pattern detection tools

Tools	Patterns	Precision	Recall
DeMIMA	Abstract Factory, Composite, Adapter, Command, Decorator, Observer, State/Strategy, Prototype, Visitor, Singleton, Template Method and Factory Method	34	100
DP-Miner	Adapter/Command, Bridge, State/Strategy and Composite	91 - 100	97
DPRE	Adapter, Bridge, Composite, Facade, Proxy and Decorator	62 - 97	-
FUJABA	All (GoF) patterns	-	-
MARRPLE	Abstract Factory, Composite and Visitor	78.6	78.3
Pinot	All (GoF) patterns except (Builder, Prototype, Command, Interpreter, Iterator and Memento)	-	
PTIDEJ	All (GoF) patterns except (Builder, Bridge, Iterator)	65	100
Tsantalis	Singleton, Composite, Adapter/Command, Decorator, Observer, State/Strategy, Prototype, Visitor, Template Method and Factory Method	100	95.9
SPQR	Decorator	-	-
WOP	Abstract Factory, Bridge, Strategy and Adapter.	57.3	54.5
DPJF	Composite, Observer, Decorator, Proxy, State, Bridge, Strategy, CoR	100	80

2.4 Code Smell Detection Approaches

Code smells can be referred as symptoms of problems that appear in a fragment of code that make software hard to maintain and change [1]. There are many approaches in the literature have been used for detection of code smells. Each approach has advantages and limitations. Several techniques are accomplished with the support of tools for code smells. Table 2.3 summarized the applied approaches with some associated tools for each approach. It shows that search and metric-based techniques are the most common techniques used by authors. In addition, search-based technique combines code metrics and machine learning methods to detect code smells. Moreover, although metrics-based code smells

detection techniques have limitations as mentioned earlier, it is applied frequently in many places. The accuracy of the approaches is very important for selecting the appropriate tool. However, it is noticeable from Table 2.3 that the accuracy of most approaches is not calculated. The literature is surveyed for code smells detection approaches. As a result, 7 categories of code smells detection approaches are presented by Kessentini et al. [44]. These techniques are explained in the following sections.

2.4.1 Manual Approach

Manual code smells detection technique is proposed by researchers in [45, 46] based on some reading guidelines. However, this approach has a human involvement, and hence it eliminates uncertainties. On the other hand, it is not effective for code smell detection for large systems.

2.4.2 Metrics-based Approach

Metrics-based detection techniques [47, 48, 49, 50, 51] are dependent on source code metrics, but they vary based on how the metrics are applied. The accuracy of this technique is dependent on the selection of metrics thresholds values. No standard values for code smells detection which result in a variance of different techniques. InCode [52] is a metric-based detection code smell tool. It has been used frequently in the literature [53, 54, 55, 56].

2.4.3 Search-based Approach

Most of the search-based techniques [57, 58, 59, 60, 61, 62] apply different machine learning algorithms for detecting code smells from source code directly. However, these algorithms learn from the standard, known design structures and then test how code deviates from the one learned before. The accuracy of this technique is dependent on the quality and size of data sets used to train the machine. Also, the detection of code smells using this technique may vary due to the variance of code smells definitions. Fontana et al. [63] presented a comparison between some machine learning-based code smells detection techniques. JDeodorant tool [16] is an example of search-based code smell detection approach. JDeodorant can only detect four code smells.

2.4.4 Symptoms-based Approach

Symptoms-based technique [64, 65, 66, 67, 68] is based on some notations which is translated to a detection algorithm. The symptoms need analysis to select the proper threshold values. The accuracy of this technique is low because of different interpretation of the same notation. Moha et al. [64] proposed a tool called Decor that depend on symptoms approach. The tool reported a high performance measures in recall rates.

2.4.5 Visualization-based Approach

Visualization-based technique [24, 69, 70] is a semi-automated method which combines human and automatic process in the detection. The technique is not effective for large systems. In addition, it is error prone because of human interaction. Simon et al. [69] presented metric-based visualization tool called Crocodile. They performed experiments on a small system. Another tool by Carneiro [70] is presented to detect God class and Divergent class code smells. The percentage of the tool were about 50%.

2.4.6 Probabilistic-based Approach

Probabilistic-based technique [71, 72] is based on fuzzy logic rules which use quantitative proprieties. Also it uses ranks for candidates of code smells using inference rules. Nitin Mathur [72] proposed a statistical analysis-based technique called Java Smell Detector (JSD). It detects only five code smells.

2.4.7 Cooperative-based Approach

Cooperative-based technique is a recent technique that combines activities in a cooperative way to perform code smell detection. After surveying the literature, only one common study presented by Abdelmoez et al. [73] applied two parallel algorithms to detect code smells. Both algorithms are based on genetic algorithms.

Table 2.3: Code Smells detection approaches

Reference year	**Tool**	**Approach**	**Languages**	**Detected Smells**	**Accuracy** (%)
[74] (2009)	JSmell	Metrics based	Java	Seven	Success rate 85-90
[50] (2010)	CodeVizard	Metrics based	Java	One	NM
[75] (2011)	Together Borland	Metrics based	Java, C#, C++	NM	NM
[76] (2012)	JCodeCanine	Metrics based	Java	Four	Average Acc 54
[77] (2013)	JSNOSE	Metrics based	Javascript	Thirteen	Rec 98 Prec 93
[78] (2014)	Research prototype	Metrics based	Java, C#	Four	NM
[47] (2015)	InCode	Metrics based	Java	Thirteen	NM
[79] (2002)	JCOSMO	Search based	Java	Three	NM
[80] (2005)	CodeNose	Search based	Java	Seven	NM
[58] (2007)	JDeodorant	Search based	Java	Four	Prec 92
[81] (2012)	BSDT	Search based	Java	Seven	Acc 100
[82] (2013)	iPlasma	Search based	Java	Four	Average Acc 90
[59] (2015)	CSD	Search based	Java	Five	Rec 58-100 Prec 72-86
[67] (2003)	LMP envirn-ment	Symptoms based	SOUL	Two	NM
[64] (2010)	Detex	Symptoms based	Java	Fifteen	Rec 100 Prec 50
[69] (2001)	Crocodile	Visualized based	Java	Four	NM
[72] (2011)	JSD	Probabilistic based	Java	Five	NM

CHAPTER 3

LITERATURE REVIEW

In this Chapter, a summary of the relevant studies in the literature that have addressed the link between DPs and code smells in terms of several aspects is demonstrated. The collected studies are based on three categories as follows:

- DPs impact on software quality attributes
- Code smells impact on software quality attributes
- The relationship between DPs and code smells

We classify the relationships between design pattern and code smell based on the following:

- **Structural** : This type of relation examines the structure of both DPs and code smells. For instance, an evaluation of specific parts of code i.e. that may make up both a design pattern and be considered as code smell.
- **Refactoring** : In this relationship, the refactoring technique is applied in order to eradicate a code smell by the use of DPs

•**Co-occurrence** : The relation means that if one of the terms i.e. DPs exist, this might affect the other term i.e. code smells to be existed.

3.1 Design Patterns Impact on Software Quality Attributes

Design patterns (DPs) are known as organized solutions to design flaws [3]. On one hand, DPs are linked to good software systems. However, applying DPs is not an easy task, in particular in case of integrating in a real software systems.

Although software engineers acknowledge the benefits of DPs, there are some studies that investigated DPs impact on software quality. This literature shows that the impact of DPs has been analyzed on few quality attributes: maintainability, change-proneness [12, 83], fault-proneness [84, 85], and performance [86], as illustrated in Figure 3.1.

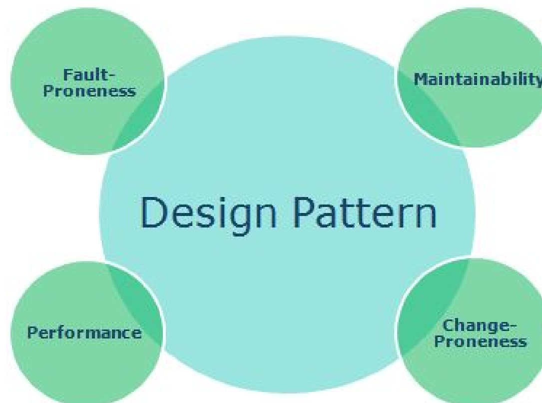


Figure 3.1: The common attributes influenced by Design Patterns

Proxy, Observer, Bridge and Command are regarded as they have a questionable use [11]. Wendorff et al. [11] stated that Proxy design pattern is easily understand-

able which consequently makes it a classical "beginner's pattern". The rationale behind the use of Proxy pattern is justified with some quality characteristics such as flexibility, access control and performance. According to the authors in [11], these quality properties never materialized in most cases. It is known that, in general, the use of design patterns tend to increase the number of classes and methods. With Proxy it is not different. Therefore, its unnecessary employment makes the size and complexity of the software increases considerably.

Wendorff [11] argues that a complicated functionality was spread over proxy classes ignoring the documentation rationale. Missing the documentation itself is not a code smell. However, when code becomes bigger and complex, the lack of documentation in this case is even worse.

Bieman et al. [12] investigated the relationships between software change proneness and design structure. Five open source systems have been used to conduct the experiment. Design patterns and class inheritance participation are used to recognize design structure. As results show in the study, classes participate in design patterns are more change prone than other classes. However, this result may be justified by the fact that design pattern provides essential key responsibility and functionality to the system. This may explain why classes that participate in design pattern are more change prone than other classes, relatively. Although it needs further investigation, change-proneness in patterns participating classes may be an indication of design patterns co-occurrences with either Divergent Change or Shotgun Surgery code smells, which are defined by Fowler et al. [1].

Prechelt et al. [5] performed an experiment which investigates software maintenance situations that use several design patterns and compares it to other scenarios with simpler alternatives. In this study, there is an interesting section right on Introduction named "Isn't This Just Obvious?". In this section, the authors state that "clearly patterns do have the advantages claimed for them!" However, our intuition may mislead us and terms such as "clearly" and "obviously" do not involve evidence. The authors find that, in many cases, the design solutions in which GoF's patterns are employed are easier to maintain than their corresponding simpler solutions. Nonetheless, the authors detach that there are situations in which the use of design patterns made maintenance of the program harder. However, these cases can be utilized as basic starting points of design patterns use guidelines. Besides that, they confirm that as compared to a straightforward solution, design patterns may provide unnecessary flexibility.

A replication study of Prechelt et al. [5] has been performed by Vokac et al. [13] and Krein et al. [87].

Vokac et al. [13] concluded that in regards to the Visitor pattern, it led to high cost in terms of development time and poor correctness. In fact, Visitor pattern has a complex structure which may justify the conclusion of study by Vokac et al. [13]. Besides, they found that Decorator eases maintenance, despite being hard to trace the control flow of the program, and hence lead to increase understandability efforts. Therefore, although its maintainability is good, it decreases understandability. Decorator also requires some training, just like Observer, but

this one may ease understandability and reduce maintenance efforts. According to the authors results, Composite pattern which relies on recursion, may cause some issues. Vokac et al. [13] justified that the rationale behind these such issues in Composite that its reliance on recursion is no longer in use. This may be caused due to the availability predefined directories and predefined containers in languages. Krein et al. [87] state that more deeper and meticulous practical analysis is needed for design pattern topics.

3.2 Code Smells Impact on Software Quality Attributes

Code smells are perceived to lead to maintenance difficulties in software systems [88, 89, 90]. In addition, some studies claim that classes which have code smells are liable to have more change prone [10, 88, 91], and more defects [88, 92, 93] than other classes that do not have code smells.

Olbrich et al. [88] have studied the impact of God classes and Brain classes on the change size, change frequency and defects. They concluded that classes having this kind of code smells have more change size, change frequency and defects. The studies [94, 95, 96, 97], line with the findings of [88]. Interestingly, Olbrich et al. [88] found that when their results are normalized with respect to size (i.e. Line of Code), the results do not hold any more under an assumption classes that are involved in code smells i.e. God Class and Brain Class have a ratio

of functionality similar to other classes on average. As they stated, code smells are not generally harmful. On opposite, such code smells in classes may be an efficient way for organizing code providing that these smelly classes are constructed intentionally. The results of [88] is consistent with the results stated in the study done by Arisholm et al. [98] of evaluating the effect of centralized control on the maintainability on large experiments of object oriented programming (OOP). Arisholm et al. [98] outlined that most of the junior developers performed better on centralized control style systems than the object oriented style versions.

D'Ambros et al. [99] conducted an empirical study on the impact of anti-patterns i.e. Blob and Spaghetti Code, on program understandability. Their conclusion outlined that only one occurrence of Blob or one occurrence of Spaghetti is not significantly decreasing the program understandability. On the other hand, a combination of both Blob and Spaghetti affects the program understandability and so affects the maintainability.

Jafaar et al. [93] reported a study on the impact of design patterns classes that have dependencies with the non-design pattern classes on the defects and change proneness. In addition, the study has been done in the case of antipattens classes and non-antipattens classes. The results of their work showed that classes having dependencies with antipattens produce more defects than the case of design patterns. However, the exhibited results in [93] may vary from one antipattern to another depending on the smell in addition to the analysed software subjects.

Foutse et al. [91] reported an experimental study concerning the impact of

antipatterns on change- and fault- proneness. They resulted that classes with antipatterns are more change prone and fault prone. In addition, they showed that structural changes affect classes participating in antipatterns more than other classes. As a summary, Figure 3.2 illustrates the common attributes that are influenced by code smells.

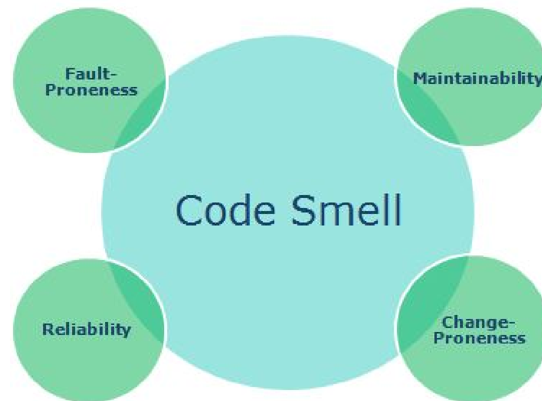


Figure 3.2: The common attributes influenced by Code Smells

3.3 The Relationship between Design Patterns and Code Smells

We detach that this is not a closed list, but just some ways to relate design patters and bad smells. We anticipate that during our analysis, we only identified studies that relate design patterns and bad smells through refactoring or structural comparison.

3.3.1 Structural Relationship

Several studies have addressed structural relationship between design patterns and code smells. Khomh et al. [100] stated that understanding of design part of a system is very effective on the quality attributes. The work proposes quality models that take into considerations several styles of the design including design patterns, code smells and anti-patterns. Authors believe that strength of design structure is important to measure the quality attributes. Therefore, they have analyzed how these styles, DPs and code smells, affect the quality attributes like fault and change proneness. Bouhours et al. [17] propose a study which aims to improve the use of design patterns. Since the designers may use DPs to get a good design and easier maintenance work, they propose a model to detect "spoiled patterns". Spoiled patterns are alternatives solutions for the same problem that design patterns may address in order to adapt the deviated problem to the traditional one solved by design pattern. Authors has stated that the spoiled patterns deteriorate the intrinsic qualities of a design pattern. Spoiled patterns are comparable with code smells.

Von Detten and Becker [101] propose an approach for reverse engineering systems. As long as the maintenance task depends on the understandability of the systems, there is a need for reverse engineering to support such understanding of a system. The approach used by the authors combines clustering-based and pattern-based techniques. The occurrence of code smells may violate the architecture issued by clustering-based approach because smelly classes tend to be

strongly coupled, and so they will be clustered together. This case is opposite to design patterns, which strive to construct a modular architecture. Interestingly, this opens our eyes to investigate whether that DPs have an influence on code smells.

In the study proposed by Polasek et al. [102], an extension of two algorithms has been introduced to detect, in addition to design patterns, anti-patterns. According to the study, anti-patterns structure is more complex than design patterns structure as anti-patterns may need several variables e.g. number of methods, name of the class, method parameters. This fact facilitates the extension of the work to detect anti patterns. Code smell and anti pattern are considered in this work as synonyms to each other. Authors state that some design patterns are source of design flaws detection. The authors analyze structural features such as associations, generalizations, and class abstraction.

3.3.2 Refactoring Relationship

This section presents studies that have addressed the relationship between DPs and code smells in terms of refactorings. Refactoring stage has a high involvement in the issue of software code quality improvement using design patterns to eliminate code smells. Several studies [14, 103, 104, 105, 106, 107] have addressed that issue.

Seng et al. [103] have discussed the importance of maintenance that helps to eliminate fragments of code which degrade the quality of systems. These frag-

ments are called code smells. In order to identify that, they proposed an automatic search-based approach to suggest the possible refactoring segments based on design patterns in the code of systems. However, applying or not applying the approach of refactoring is a user-decision. However, the impact of refactorings based on design patterns which called "Refactoring to Patterns" has been empirically assessed by Alshayeb [108]. He found that there is no clear trends about refactoring and refactoring to patterns in quality improvements.

Trifu and Reupke [104] worked on the issue of software maintenance practices and the cost it involves. According to their perspectives, maintenance tasks may exist due to two factors: poor design and the age of the system. However, they introduced a tool that suggests the needed refactorings to the structural flaws in OOP systems. Design patters are suggested to be applied as part of the refactorings to remove code smells.

Jebelean et al. [14] proposed an approach that detects code smells automatically and list the parts of code to be refactored. Composite design pattern is involved in their approach as a refactoring suggestion. The purpose of applying composite design pattern is to compose objects into tree structures.

Fontana and Spinelli [106] have analysed how the refactorings can affect the software evolution and maintenance using metrics approach to assess the quality. Two design patterns, Visitor and Strategy, are cited in the study to have an effect on removing the bad smell Divergent Change. A co-occurrence between these patterns and the bad smell Feature Envy may exist. These findings can bring

researchers' attention to the issue of co-occurrence of DP and bad smell. Dorman and Rajlich [107] reported the improvement experience done by a programmer to an open source system. As part of these improvements, the authors have discussed sort of relationship between the elimination of (bad smells and changes) and the use of design patterns. Refactorings are treated as a stage in the software change process. However, applying refactoring tasks does not improve the quality attributes necessarily [109].

3.3.3 Co-occurrence Relationship

Walter et al. [36] conducted an empirical study to evaluate the relationship between DPs and code smells. This work addressed the high level (i.e. design level) to study such relation. On the other hand, the authors have examined this relation in a pattern level for 10 design patterns and 7 code smells only. Their experiments are mainly done based on two major open source projects with some subsequent releases of each project. The results indicate that the presence of DPs is not strongly linked with code smell instances. However, these observations become more supported for some patterns such as Singleton and less supported for others such as Composite.

As a summary, Figure 3.3 illustrates the common relations between design patterns and code smells.

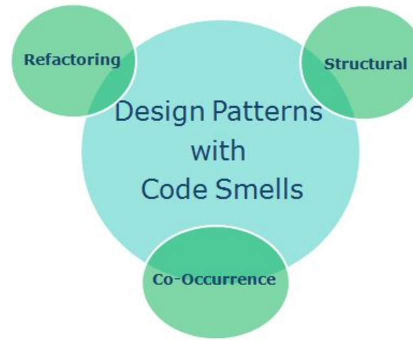


Figure 3.3: The common relations between Design Patterns and Code Smells

3.4 Summary

Design patterns are not generally good or generally bad. They have to be used in an appropriate manner to match an appropriate problem. For instance, a documented design patterns with essential training can improve maintenance tasks in terms of time and quality properties. Based on the literature conducted in this study we have extracted the design quality attributes commonly used in the studies. The previous section presented some works related to design patterns and bad smells. As noticed, some papers focused on design patterns and software quality attributes. Others exhibited code smells and software quality attributes. Interestingly, some works analyzed the potential problems of design patterns and their consequences. For instance, some problematic use of design patterns, similar to Proxy, Observer, Bridge, and Command is identified by Wendorff [11]. After the performed literature review, it was found that most of the studies focus on refactoring opportunities. On the other hand, some studies establish structural analysis concerning design patterns and bad smells. In the sequence, we narrowed our research to evaluate the relationship between design patterns and code

smells. By accomplishing this literature review, we realized that the co-occurrence between design patterns and bad smells is unexplored theme, although it is mentioned in some papers. The objective of this work is to empirically evaluate the effectiveness of design patterns on the code bad smells. Consequently, this work identifies the points at which design patterns co-occur with bad smells.

CHAPTER 4

EXPIREMENTAL SETUP

This chapter describes the data used for conducting the study. The goal of this work is to empirically evaluate the impact of the participating design pattern classes on the smell presence. Data collection, methodology plan and then the measurement tests are presented in this chapter.

4.1 Data Collection

In order to conduct this work, 10 open source software systems have been collected. Hence, the required data set is as follows: (1) Classes which are participating in patterns, (2) Classes which are participating in code smells.

4.1.1 Subject Systems

There follows a description of the selected subject systems: DrJava v20020804, JHotDraw v5.1, DrJava v20020619, DrJava v20020703, MapperXML v1.9.7, Nutch v0.4, PMD v1.8, JUnit v3.7, QuickUML 2001, Lexi v0.1.1.

- **DrJava:** The Dr.Java project provides a user friendly, simplified environment for writing and building Java programs. It includes the following release versions: **DrJava v20020804, DrJava v20020619, DrJava v20020703**. The project allows users to assess Java code in an interactive environment and it includes an intuitive interface that was created with students in mind.

- **JHotDraw v5.1:** A framework that supports the design of drawing editors. Numerous tasks are facilitated in the editor including the ability to devise behavioural constraints, design and edit geometric and user defined shapes, and create animation.

- **MapperXML v1.9.7:** MapperXML v1.9.7 can be utilized for web applications as a presentation framework. It functions by creating applications through different components which conform to the Model-View-Controller design.

- **Nutch v0.4:** Nutch v0.4 is a scalable web crawler which is also extensible. Among its useful functions are searching for, indexing and scoring filers.

- **PMD v1.8:** PMD v1.8 is a source code analyzer which can detect standard coding rule abuses by scanning the source code. It is also useful for locating dead and suboptimal code.

- **JUnit v3.7:** JUnit v3.7 is a valuable tool for writing test cases for Java programs. It is known for its simplicity of design which allows the user to

create test cases which can be used repetitively.

- **QuickUML:** QuickUML is an object-oriented design tool which is extremely user friendly. It facilitates very comprehensive integration of a core set of UML models. It enables the user to access their complete project by utilizing a tabbed window with smooth integration between object models, use cases, class models, dictionaries , and code.

- **Lexi v0.1.1:** A word processor that enables editing of plain text as well as numerous file types like RTF and HTML.

4.1.2 Data of Patterns and Smells

The reason for choosing the already described systems is the availability of pattern data in the P-Mart repository [110]. As explained in Chapter 2, the pattern identification and detection tools are not suitable for our work since that all of them are not congruent with the characteristics required to obtain reliable data. As a result of our search, **P-Mart repository** came across. Several works [40, 41, 42, 43] have used it due to its reliable source and validation.

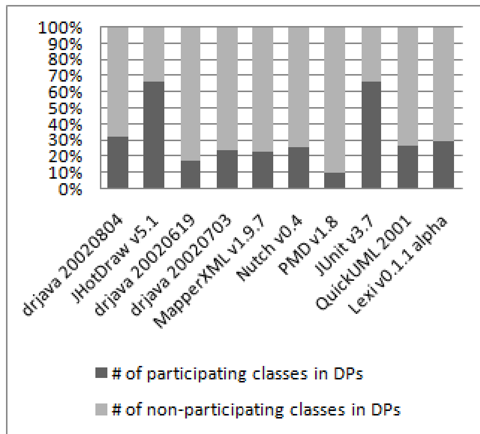
P-Mart repository was collected based on GoF's book [3]. It consists of 10 systems. Each system patterns collected in a separate session by B.Sc. and M.Sc. students. Also, the collections of other people who are working in this area are used as well.

Table 4.1 provides several statistics for the subject systems whilst Table 4.2 reports the number of patterns instances in each one of the subject systems. We can

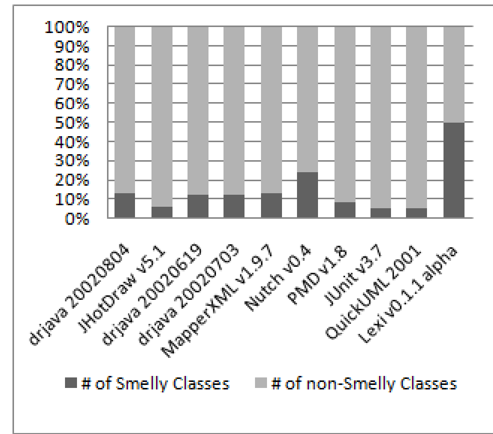
Table 4.1: Projects statistics

Systems	# classes	LOC	Code Lines	# Smelly Classes	# DP classes
drJava 20020804	267	61844	30390	(13%)34	(32%)85
JHotDraw v5.1	155	16085	8891	(5%)8	(66%)103
drJava 20020619	215	47617	22696	(11%)25	(19%)41
drJava 20020703	238	52870	25425	(12%)28	(23%)55
MapperXML v1.9.7	217	32667	14372	(12%)27	(22%)48
Nutch v0.4	165	37106	23507	(24%)39	(25%)41
PMD v1.8	446	52302	41321	(8%)35	(9%)43
JUnit v3.7	78	6517	4773	(5%)4	(67%)52
QuickUML 2001	156	23319	9249	(4%)7	(26%)41
Lexi v0.1.1 alpha	24	10005	7045	(50%)12	(29%)7
Total # (All Systems)	**1961**	**340332**	**187669**	**219**	**516**

see in these tables that the number of DPs and the percentages of the participating classes in DPs are distributed over the systems. Patterns micro architecture instances range from 5 to 34 over the subject systems, as seen in Table 4.2. In Table 4.1, design pattern participant classes approximately range from 9% to 66%. The same thing can be said about the number and the percentages of smelly classes in the subject systems. The percentage of smelly classes ranges from (5%) to (50%) in the subject classes. Figure 4.1a and Figure 4.1b clarify these numbers.



(a) DPs participant to non-participant classes



(b) Smells participant to non-participant classes

Figure 4.1: Participant to non-participant classes

Table 4.2: Patterns instances in the subject systems

Category	pattern	drJava-20020804	JHotDraw v5.1	drJava-20020619	drJava-20020703	MapperXML v1.9.7	Nutch v0.4	PMD v1.8	JUnit v3.7	QuickUML	Lexi v0.1.1
Creational Patterns	Abs Factory					1				1	
	Builder				1			2		1	1
	Factory Method	1	3		1	1		3			
	Prototype		2								
	Singleton	8	2	8		3	1		2	1	2
Structural patterns	Adapter	2	1	2	1	2	2	1			
	Bridge	1					2				
	Composite		1			1		2	1	1	
	Decorator		1						1		
	Facade					1					
	Flyweight										
	Proxy	1		1	1			1			
Behavioral patterns	Chain of Resp.										
	Command	2	1	2			2			1	
	Interpreter										
	Iterator	1		1			1	1	1		
	Mediator	1									
	Memento	1		1			2				
	Observer		2			1		2	3	1	2
	State	3	2		3						
	Strategy	3	4	1	1	1	2				
	Template	9	2		8	4	3	1			
	Visitor	1		1				1			
SUM		{34}	{21}	{17}	{16}	{15}	{15}	{14}	{8}	{6}	{5}

The comparison of code smells detection tools has an impact on conducting the study. Some factors in the literature have been exposed to be used in the analysis of code smells detection tools. Evaluation the accuracy of the tool is an important factor for the selection of an appropriate code smell detection tool. Precision and recall are factors to identify correctness and completeness of results as described in Chapter 2. Source code metrics-based tools are applied commonly. According to the literature, none of the tool detects all the 22 code smells by Fowler [1]. InFusion tool [111] which is the extended version of InCode [52] tool (while iPlasma [112] is the old version of InCode) can detect 22 code smells, 10 of them are identified by Fowler [1]. Since the accuracy of iPlasma tool at average was 90% and InFusion is the newer version, this may increase the confidence of InFusion tool. InCode, the old version of InFusion has been frequently used in the literature studies. Moreover, InFusion provides well-documented definitions of the detection rules and techniques used and their associated metrics with references.

In addition, InFusion makes quality assurance of multi-million LOC systems not merely practical, but effective, successfully handling both object oriented and procedural style code. Whether you own, are responsible for, or are acquiring software projects in C/C++ or Java, InFusion puts you in full control of architecture and design quality. It supports visualization and refactorings.

Moreover, InFusion classifies the smells' effectiveness on the quality attributes e.g. complexity. It provides some metrics for quality e.g quality deficit value for separated classes and methods. The function level of detection is provided by

InFusion tool. We collected smells data using InFusion tool. As mentioned before, the subject systems of this study are open-source systems. They are hosted in the sourceforge website[1].

4.2 Methodology Description

In order to achieve the objectives we planned, we have to decide on how to measure the differences between groups i.e. design pattern group vs non-design pattern group. We followed a set of phases. In general, the methodology in this work consists of three phases as illustrated in Figure 4.2.

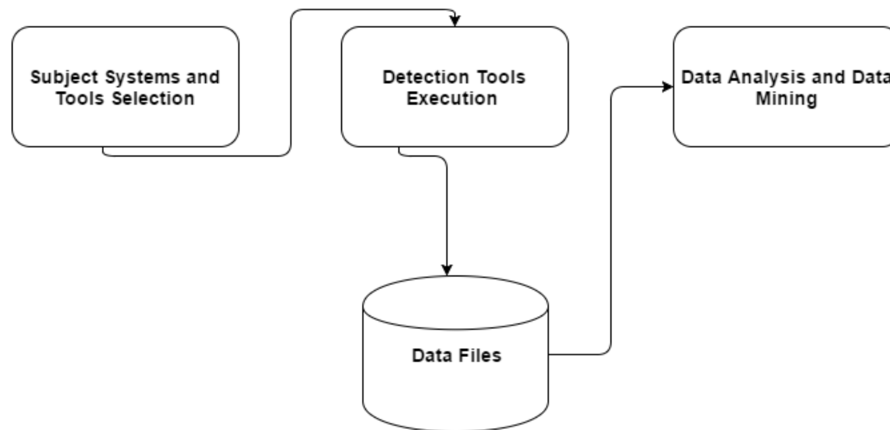


Figure 4.2: Methodology flow

The phases are: (1) Subject systems and detection tools selection. (2) Execution of the tools to bring the related results of design patterns and code smells data. (3) Data analysis and mining the data. The first phase is **Subject Systems and**

[1]https://www.sourceforge.net

Tools Selection. This phase brings us the systems to be analyzed in the work. Beside choosing the systems, it focuses on selecting design patterns and code smells tools. As discussed previously in Section 4.1.2, only code smells detection tool was chosen while P-mart repository was used for design pattern data. The parameters configuration of the code smells detection tool remained with the default values. The second phase is **Detection Tools Execution**. The main point in this stage is to collect code smells and design patterns data of each system. As discussed before, for the design patterns we used the xml files in the P-mart repository and parsed it to get each class in the xml file and its corresponding design pattern. For code smells, we ran the tool for each system and collect the smelly classes along with the number of smells in each class with their types. The previous output results of design patterns and code smells were stored in the created data files (The database in the Figure 4.2). The files were created as one file per project. Each file has 25 columns. First column is the class name followed by the occurrences of a design pattern. Then, the column number three indicates code smells occurrences followed by the column of the number of smells in that class. The column after that specifies the type of smells the class may have. The remaining 20 columns refer to the corresponding design pattern types classified based on the categories of design patterns. The last phase is **Data Analysis and Mining**. The purpose in this stage is to analyze and mine the data in the stored files.

Statistical measurements were identified to achieve the objectives as follows:

1. **Objective 1 - Empirical evaluation of code smell-proneness and**

smell density in design pattern classes versus non-design pattern classes: For measuring the differences between the groups i.e. design patterns and non-design patterns, we chose to use Wilcoxon signed-rank test [113] and OR test [114] in order to compare the significance of the overall data. The results are shown in Section 5.1.1.

2. **Objective 2 - Empirical evaluation of code smell-proneness in design pattern classes versus non-design pattern classes in each category of the design patterns(creational, structural and behavioral)**: Mann-Whitney U test [115] is used in order to do comparison between pairs of data, whilst Kruskal Wallis test [116] is dedicated for measuring multiple groups(categories) of data. The results are shown in Section 5.1.2.

3. **Objective 3 - Empirical evaluation of code smell-proneness in design pattern classes versus non-design pattern classes for the individual design patterns**: To achieve the goal in this point, we split the stage into two steps: (1) The comparison among patterns in each category using Kruskal Wallis test. (2) The association rules analysis using Apriori algorithm. The results are shown in Section 5.1.3.

The independent variables in this work are the classes participating in the design patterns and the dependent variables are the smelly classes. Next follows have detailed descriptions of the identified statistical measurements in this work.

Table 4.3: Events involved in OR test

DP \Smell	**Yes**	**NO**
Yes	1	2
NO	3	4

4.2.1 Odd Ratio (OR) Test

We calculated the odds ratio (OR) [114] which specifies the probability of an event occurring. The odds ratio can be calculated based on two groups. The first group is the design patterns sample (p), while the other group is the code smells sample (q). The ratio is given as follows: $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 points to the equality of the both samples. This indicates that the occurrence is equally probable in both samples, while OR of points greater than 1 means that the first sample in the numerator is more likely to have smells. On the other hand, OR value of points less than 1 indicates that the second sample has more probable of smells. This is typically used when the sample size is small. Four events are involved in this test: (1) Classes participating in design patterns and code smells. (2) Classes participating in design patterns only. (3) Classes participating in code smells only. (4) Classes not participating in any of design patterns and code smells. Table 4.3 clarifies these events.

4.2.2 Wilcoxon Signed-Rank Test

Wilcoxon signed-rank test [113] is well known, non-parametric test used to compare differences between two paired data. It is used as a powerful replacement for t-test which is used for normal distribution. It is an effective replacement but

slightly less powerful than t-test. The variables in this context are considered as continuous.

4.2.3 Mann-Whitney U Test

This test is utilized to evaluate the variations between different groups by making a comparison between two independent groups [115]. This non-parametric test can facilitate a comparison between classes that participate in design patterns and the non-participating classes in terms of the disparity in smell-proneness. For instance, the use of this test is to assess the variation between the classes that participate in the structural design patterns and the classes that participate in the creational design patterns, in terms of smell proneness. In order to conduct this test the following four assumptions must apply:

- Dependent variables are required to be one of the following: categorical or continuous. For the case being examined, the values of smell-proneness are categorical (either 0 or 1).

- It is important to organize independent variables to be as two groups in a shape of categories. The underwent test values have 0 or 1. 0 corresponds to the group of non-participating classes while 1 corresponds to the group of participating classes.

- It is vital that the independence of observations is ensured. In the case being studied, the condition applies.

Table 4.4: Test of normality

Tested Variable	df	Group	**Kolmogorov-Smirnov Sig value**	**Shapiro-Wilk Sig value**
Smell-Proneness	1961	Participant	**0.000**	**0.000**
		Non-Participant	**0.000**	**0.000**
Smell-Density	1961	Participant	**0.000**	**0.000**
		Non-Participant	**0.000**	**0.000**

- The data should not be normally distributed. In order to test such a condition, two techniques were applied, Kolmogorov-Smirnov and Shapiro-Wilk. The results associated with the tests are presenetd in Table 4.4. When assessing the normality, the following assumptions are made:

  - H0: the data are not normally distributed.
  - H1: the data are normally distributed.

  The p-value associated with evaluating the normality was found to be less than 0.05. For this reason, it was not possible to reject the null hypothesis. This lead to accepting the H0 Hypothesis. That means the data of the groups are not normally distributed.

- The groups underwent a test should have identical distributions. Levene's test is an inferential statistic used to assess the equality of variances for a variable calculated for two or more groups [117]. we found that only the data of the design pattern participant group have identical distribution as shown in Table 4.5. The p-value of Levene test is 0.057 .

Table 4.5: Test of homogeneity of variance

<table>
<tr><th>Design Pattern Category</th><th>Skewness</th><th>Levene Statistic</th><th>p-value - Sig.</th></tr>
<tr><td>Creational</td><td>2.644</td><td rowspan="3">2.883</td><td rowspan="3">0.057</td></tr>
<tr><td>Structural</td><td>3.804</td></tr>
<tr><td>Behavioural</td><td>2.294</td></tr>
</table>

4.2.4 Kruskal-Wallis Test

The Kruskal-Wallis test [116] is utilized to make a comparison among multiple groups. In order to conduct this test, the following two assumptions must be in place:

- Dependent variables are required to be either continuous or categorical. In the case under investigation, the values for smell-proneness are categorical (either 0 or 1).

- It is necessary for the independent variables to be composed of multiple groups having categorical cases. In the case under investigation the values for patterns are categorical (either 0 or 1). In the context of this work, it is used to compare the differences among design patterns of each category.

CHAPTER 5

EXPERIMENTAL RESULTS

The purpose of this chapter is to quantitatively evaluate the functional effectiveness validity of design patterns on code smells. Before proceeding with that, the descriptive statistics for smell-proneness and smell-density are presented in Table 5.1. It is clear that the minimum value in all metrics of all cases is 0. Also, we can see that the maximum value for smell-proneness is 1 in all cases. For the other statistics, they are different from one case to another. From now on wards, we abbreviate smelly classes participating in design pattern as SDP. On the other hand, we use SnDP as an abbreviation of smelly classes not participating in design patterns. In addition, S is used for Smells.

Table 5.1: Descriptive statistics of smells in each system

Metric	Minimum	Maximum	Mean	Std. Dev
QuickUML				
Smell-Proneness	.00	1.00	.0486	.21580
Smell-Density	.00	4.00	.0641	.37121
Lexi				
Smell-Proneness	.00	1.00	.5000	.51075
Smell-Density	.00	6.00	.8750	1.51263
JUnit				
Smell-Proneness	.00	1.00	.0513	.22200
Smell-Density	.00	1.00	.0513	.22200
JHotDraw				
Smell-Proneness	.00	1.00	.0516	.22196
Smell-Density	.00	3.00	.0710	.34396
MapperXML				
Smell-Proneness	.00	1.00	.1244	.33083
Smell-Density	.00	5.00	.1613	.54158
Nutch				
Smell-Proneness	.00	1.00	.2364	.42614
Smell-Density	.00	3.00	.3939	.79401
PMD				
Smell-Proneness	.00	1.00	.0785	.26922
Smell-Density	.00	34.00	.2063	1.69800
DrJava v2002619				
Smell-Proneness	.00	1.00	.1163	.32131
Smell-Density	.00	4.00	.1721	.54956
DrJava v2002703				
Smell-Proneness	.00	1.00	.1176	.32287
Smell-Density	.00	4.00	.1723	.54322
DrJava v2002804				
Smell-Proneness	.00	1.00	.1273	.33398
Smell-Density	.00	4.00	.1873	.56408
All Systems				
Smell-Proneness	.00	1.00	.1117	.31505
Smell-Density	.00	34.00	.1866	.95557

5.1 Functional Effectiveness Evaluation of Design Patterns on Code Smells

The evaluation in this study is conducted based on three levels of design patterns. In this part, Section 5.1.1 is presented to achieve the objective number one. To do so, OR test and Wilcoxon test are mainly chosen to evaluate the relation between design patterns and code smells in the Class Level. Next follows is Section 5.1.2 which presents the results after evaluating the relation between design patterns and code smells in the Category Level. Kruskal Wallis and Man Whitney U test are selected to achieve the objective number two. The last objective of this study is accomplished in Section 5.1.3. The work in this section is divided into two steps: the first one is the comparison among patterns in each category using Kruskal Wallis test. The other step is association rules analysis using Apriori algorithm.

5.1.1 Design Level(Class Level)

RQ1: Are design pattern classes more smell-prone than the non-design pattern classes in software systems?

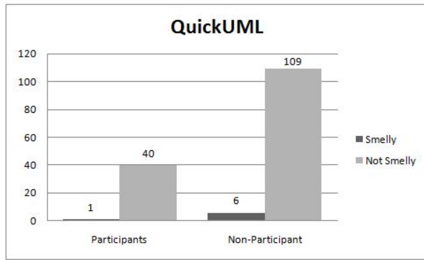
In order to answer RQ1, we evaluate the difference between participant and non-participant design pattern classes with respect to smell proneness and smell density. This evaluation provides a superficial insight on the impact of design patterns on presence and density of smells.

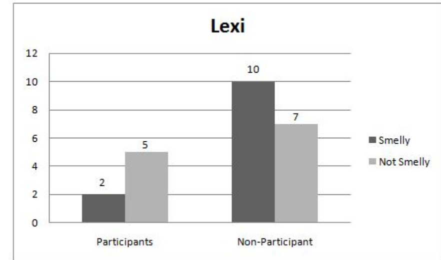
5.1.1.1 Smell Proneness(Odd Ratio Test)

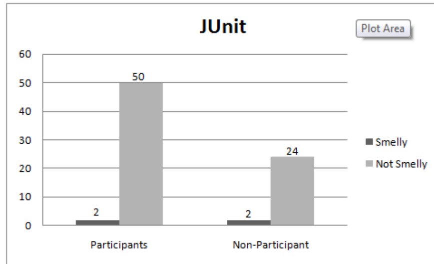
The evaluation results of smell-proneness in participating versus non-participating design pattern classes using Odd Ratio test are shown in Table 5.2. As noticeable from the table, most of the subject systems show significant differences i.e the event of smells are more likely associated with the group of non-design pattern classes. Moreover, when all systems are combined, we found that the results still hold. Only two values: Nutch system and PMD system have opposite results. Nutch system comprises 24.8% as design pattern classes while PMD system covers only 9.6%. Link to Table 5.2, the OR value of PMD system is more than 2. This might interpret the results of PMD system. Moreover, we can see from Figure 5.1 that the non-participating design pattern classes group is more likely to have a smell event than the participating design pattern group in most of the subject systems. The case is opposite in only JUnit and JHotDraw systems although the OR values of them are less than 1. In general, the data needs more investigation and analysis. Next section shows more rigorous analysis of the data.

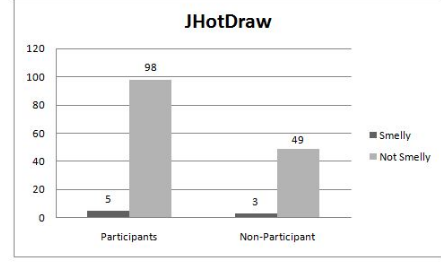
Table 5.2: Odd Ratio test analysis for smell-proneness evaluation of participant vs. non-participant design pattern classes groups

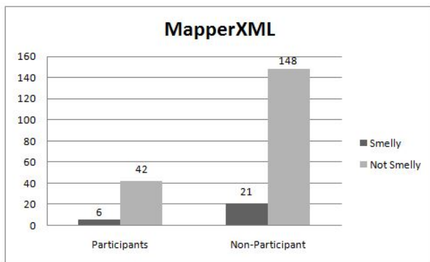
QuickUML	**.454**	<1
Lexi	**.280**	<1
JUnit	**.480**	<1
JHotDraw	**.833**	<1
MapperXML	1.007	≈ 1
Nutch	1.750	>1
PMD	2.091	>1
DrJava v2002619	1.069	≈ 1
DrJava v2002703	**.696**	<1
DrJava v2002804	**.744**	<1
All systems	**.907**	$\mathbf{\underline{<1}}$

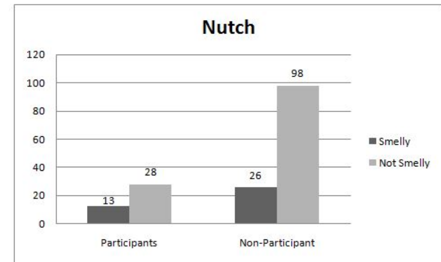

QuickUML
120
100
80
60
40
20
0
1
40
6
109
Participants
Non-Participant
Smelly
Not Smelly

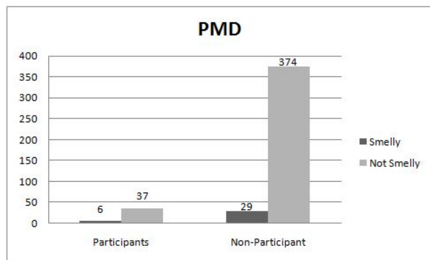

Lexi
12
10
8
6
4
2
0
2
5
10
7
Participants
Non-Participant
Smelly
Not Smelly

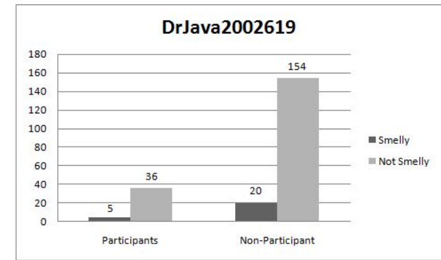

JUnit
Plot Area
60
50
40
30
20
10
0
2
50
2
24
Participants
Non-Participant
Smelly
Not Smelly

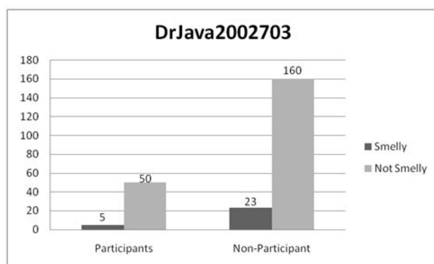

JHotDraw
120
100
80
60
40
20
0
5
98
3
49
Participants
Non-Participant
Smelly
Not Smelly

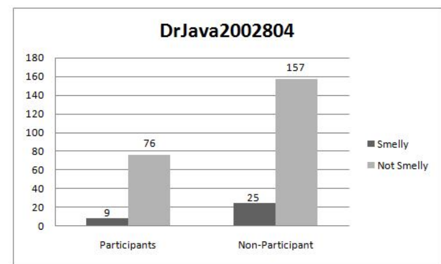

MapperXML
160
140
120
100
80
60
40
20
0
6
42
21
148
Participants
Non-Participant
Smelly
Not Smelly


Nutch
120
100
80
60
40
20
0
13
28
26
98
Participants
Non-Participant
Smelly
Not Smelly


PMD
400
350
300
250
200
150
100
50
0
6
37
29
374
Participants
Non-Participant
Smelly
Not Smelly


DrJava2002619
180
160
140
120
100
80
60
40
20
0
5
36
20
154
Participants
Non-Participant
Smelly
Not Smelly


DrJava2002703
180
160
140
120
100
80
60
40
20
0
5
50
23
160
Participants
Non-Participant
Smelly
Not Smelly


DrJava2002804
180
160
140
120
100
80
60
40
20
0
9
76
25
157
Participants
Non-Participant
Smelly
Not Smelly

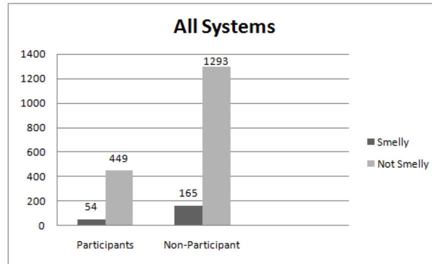


Figure 5.1: Smell proneness comparison of participant vs. non-participant design pattern classes in all systems

5.1.1.2 Smell Proneness(Wilcoxon Test)

Given the fact about the data distribution of samples: SDP and SnDP in Table 4.4 on page 54, we proceeded to conduct more strict and powerful tests. From the inspection of mean and median values shown in Table 5.3, we can visually expect that the design pattern instances are smaller as presented in the Figure 5.2. However, this expectation needs more verification. Wilcoxon test is used as an effective and powerful alternative for t-test, used for normal-distributed samples. Due to the characteristics of Wilcoxon test and the fact that combining smelly design pattern and smelly non-design pattern classes represent all smells, we linked each system with two values: SDP/S and SnDP/S , where S represents all smells in each subject systems. The results are as follows: (**z = -2.547, p-value=0.011**). Hence, the classes which participate in design patterns are less smell-proneness than the classes not participating in design patterns for the subject systems, at 95% confidence level.

Table 5.3: Statistics of smell proneness in all systems

Metric	**SDP/S**	**SnDP/S**
Mean	.280	.719
Median	.211	.788
Std. Dev.	.161	.161
Variance	.026	.026

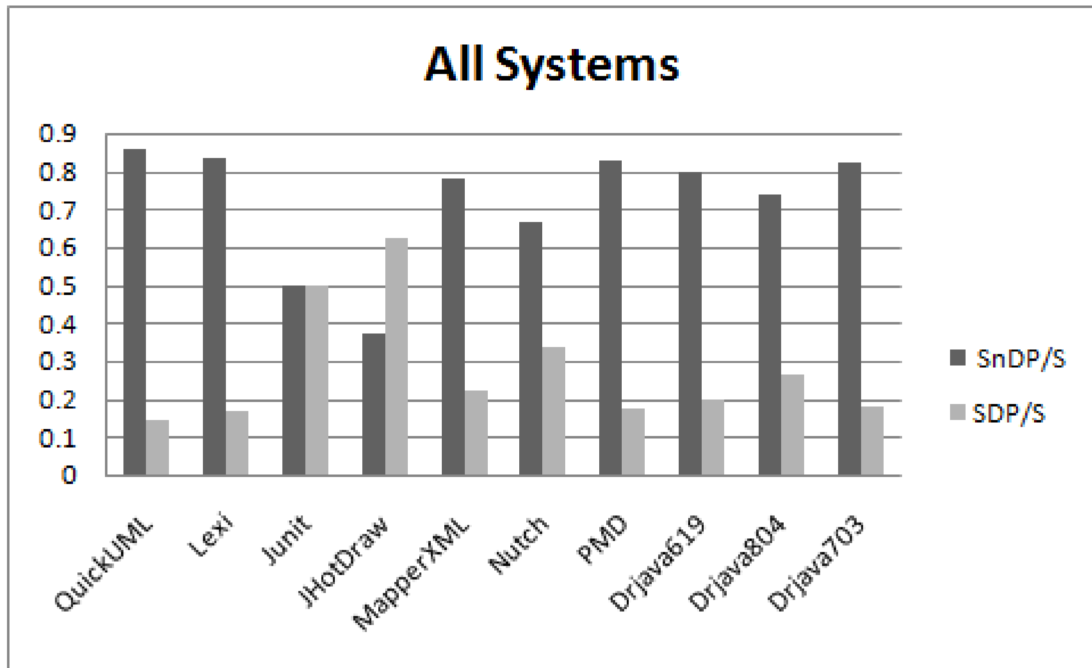


Figure 5.2: The respective values of smell proneness in SDP and SnDP

5.1.1.3 Smell Density(Odd Ratio Test)

Odd Ratio test cannot be computed on smell density sample due to the nature of this test. It only takes samples of kind 2*2 table i.e a sample underwent OR test should have four and only four cases. They are as follows:

- Design patterns and smells.

  Design patterns and non-smells.

- Non-design patterns and smells.

  Non-design patterns and non-smells.

5.1.1.4 Smell Density(Wilcoxon Test)

Similar to the smell proneness, we conducted the same procedure for smell density. Smell density indicates the number of smells associated per each class in both groups, SDP and SnDP. The resulting values are shown in diagram 5.3. The respective mean and median values are represented in Table 5.4. As long as the sample is more confidence when all subject systems are combined, we did the test based on the all systems. The results are as follows: (**z = -2.310, p-value=0.021**). Consequently, the classes which participate in design patterns are less smell-dense than classes not participating in design patterns, for the subject systems, at 95% confidence level. Interestingly, the results of smell proneness and smell density are consisted. Hence, our next experiments are smell proneness-based and not smell density-based. The justification is that most of the classes in the subject systems have only one smell.

Table 5.4: Statistics of smell density in all systems

Metric	**SDP/S**	**SnDP/S**
Mean	.297	.703
Median	.238	.762
Std. Dev.	.190	.190
Variance	.036	.036

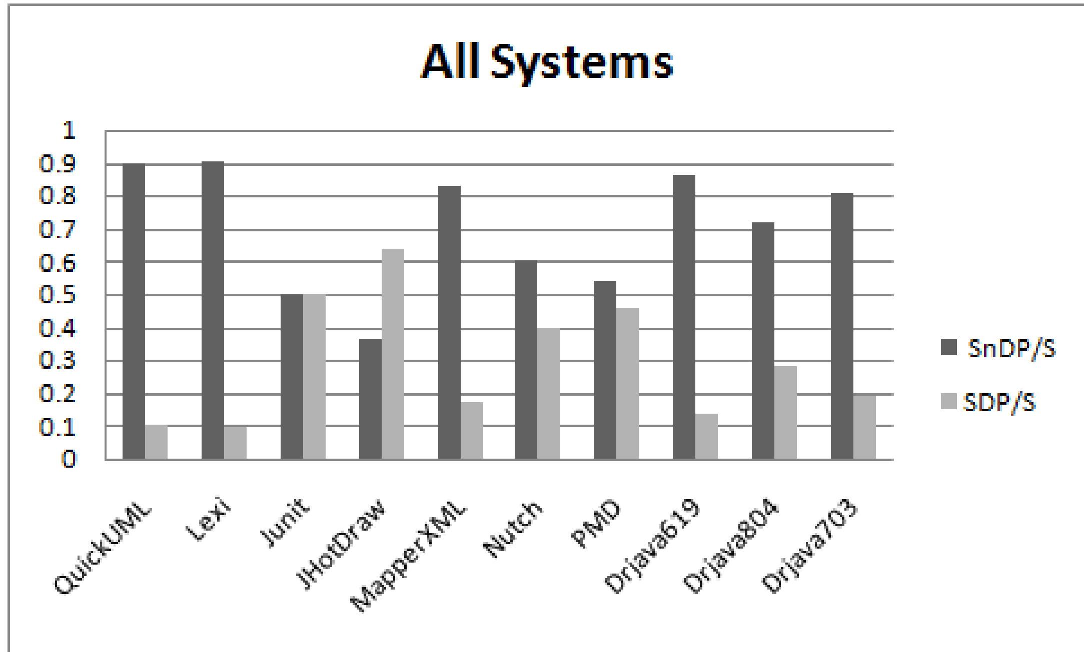


Figure 5.3: The respective values of smell density in SDP and SnDP

5.1.2 Category Level

RQ2: Do code smells have significant differences when they present in the different categories of design pattern classes?

There exist three categories for design patterns in the GOF book: Creational, Structural and Behavioural. In order to answer RQ2, we evaluate the differences in smell-proneness among these different categories. To do so, we identified 3 pairs to be underwent statistical test. Prior to do comparisons between the pairs, we conducted **Kruskal-Wallis** test in order to find whether a difference among the categories exists or not. If there is a difference, we will go further and conduct **Mann-Whitney** test in order to compare the pairs. However, we did both tests to confirm the results. The pairs in our study are as follows:

1. Creational vs. Structural

2. Creational vs. Behavioural

3. Structural vs. Behavioural

5.1.2.1 Evaluation of different design pattern categories(Kruskal Wallis Test)

In evaluating the differences in functional effectiveness of design pattern on smell proneness among the classes that participate in the different design pattern categories, we ignored the Nutch, JUnit and DrJava v2002703 subject systems. This is due to that they are only associated with 1, 2 and 2 classes of creational category, respectively. Also, we ignored Lexi because the number of the classes that participate in the structural design patterns is zero. So, we end up with 7 cases only including the all systems case.

- Smell Proneness(Kruskal-Wallis Test)

  As shown in Table 5.5 and visually inspection from Figure 5.4, the p-values obtained in this test have no significant values in all systems, even when all systems are combined, at 95% confidence level. Noticeably, JHotDraw and DrJava v2002804 systems have very close percentages of smelly design pattern classes in the categories with a maximum of 3.8% at the behavioural category and minimum of 2.2% at the structural category for JHotDraw system, and for DrJava v2002804, the percentage values of the categories: creational, structural and behavioural are 9.1%, 11.4% and 12.5% respectively. QuickUML and DrJava v2002619 systems have no smells linked to

the structural design pattern category. This observation might be due to the nature and structure of the structural category. According to [85], the structural category tends to be less prone and less change. Moreover, based on the authors' teaching experience in the field of object oriented design patterns, students may have more understanding of structural category as compared to other categories, and so more easy to apply properly. PMD and MapperXML systems have numerous variations of smells in the categories. Interestingly, when all systems are combined, the categories have almost the same values of smelly design patterns with 8.9% in all categories. This observation might indicate that design pattern categories result in the same level of reliable software.

Table 5.5: P-values of evaluation design pattern categories using Kruskal-Wallis test

Systems	**p-value**
QuickUML	.417
JHotDraw	.872
MapperXML	.333
PMD	.338
DrJava v2002619	.081
DrJava v2002804	.986
All Systems	.987

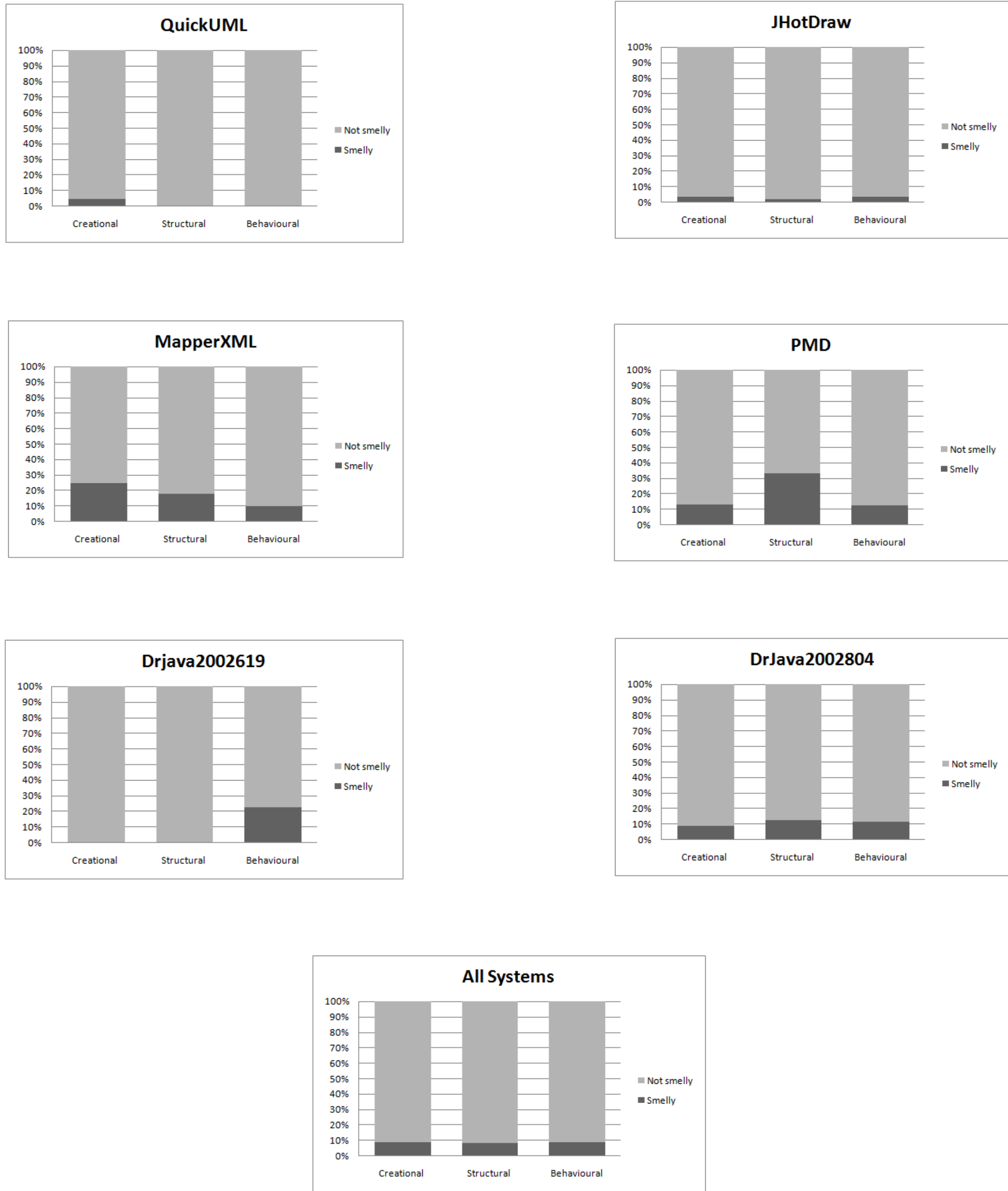


Figure 5.4: Smell proneness comparison of the design patterns categories

Table 5.6: P-values of evaluation design pattern categories using Mann Whitney test

System \Pair	**Creational vs Structural**	**Creational vs Behavioural**	**Structural vs Behavioural**
QuickUML	.386	.317	1.000
JHotDraw	.657	.967	.612
MapperXML	.552	.139	.372
PMD	.193	.961	.221
DrJava v2002619	1.000	.146	.078
DrJava v2002804	.871	.894	.929
All Systems	.907	.976	.874

5.1.2.2 Evaluation the pairs of design pattern categories(Mann Whitney Test)

Given the fact about the negligible differences in smell proneness among the design pattern categories, we do not need to proceed with Mann Whitney test. However, we conducted Mann Whitney test for pairs of categories mentioned in Section 5.1.2 to strengthen our results. P-values obtained for the pairs are shown in Table 5.6. As noticeable, the results have no significant p-values in all pairs. Consequently, design patterns categories have no significant differences in terms of smell proneness for the subject systems. This observation confirms the reported results in the previous section, i.e. the adoption of any design pattern category might produce the same reliable software.

5.1.3 Individual Design Patterns Level

RQ3: Are the participant classes in a specific individual design pattern more smell-prone in specific smells than other ones?

To answer RQ3, this section evaluates the functional effectiveness of design patterns on code smells in the individual patterns level, as follows:

- The differences in smell proneness among overall design patterns in each category. To do so, we use Kruskal Wallis test.
- If we find a significant differences in the previous test, we will go further to evaluate the co-occurrence of each design pattern-code smell pair using association rules in the Apriori algorithm.

In this section, we conducted the evaluation when all systems are combined. The reason behind it, is that the subject systems do not have the same set of patterns. The comparison of smell proneness in each category (creatinal, structural and behavioural) is presented in the Figures 5.5, 5.6, 5.7, respectively.

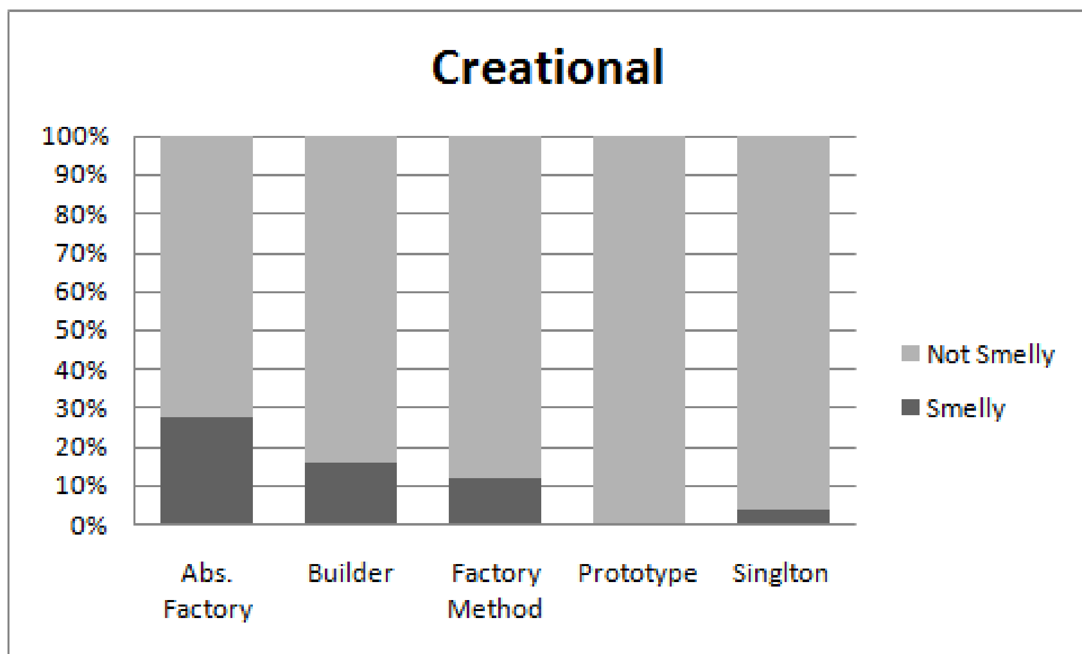


Figure 5.5: Comparison of smell proneness in creational category

Moreover, we used Kruskal Wallis technique to test the significant differences among patterns in each category, as shown in Table 5.7.

It is noticeable from Table 5.7 that each category has significant p-value which

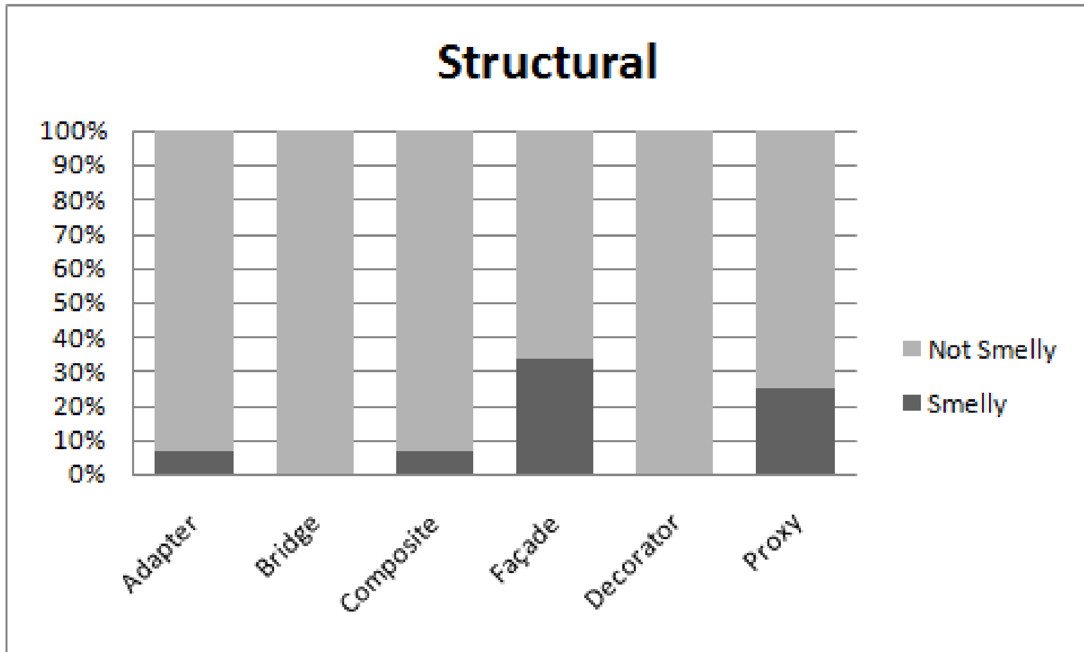


Figure 5.6: Comparison of smell proneness in structural category

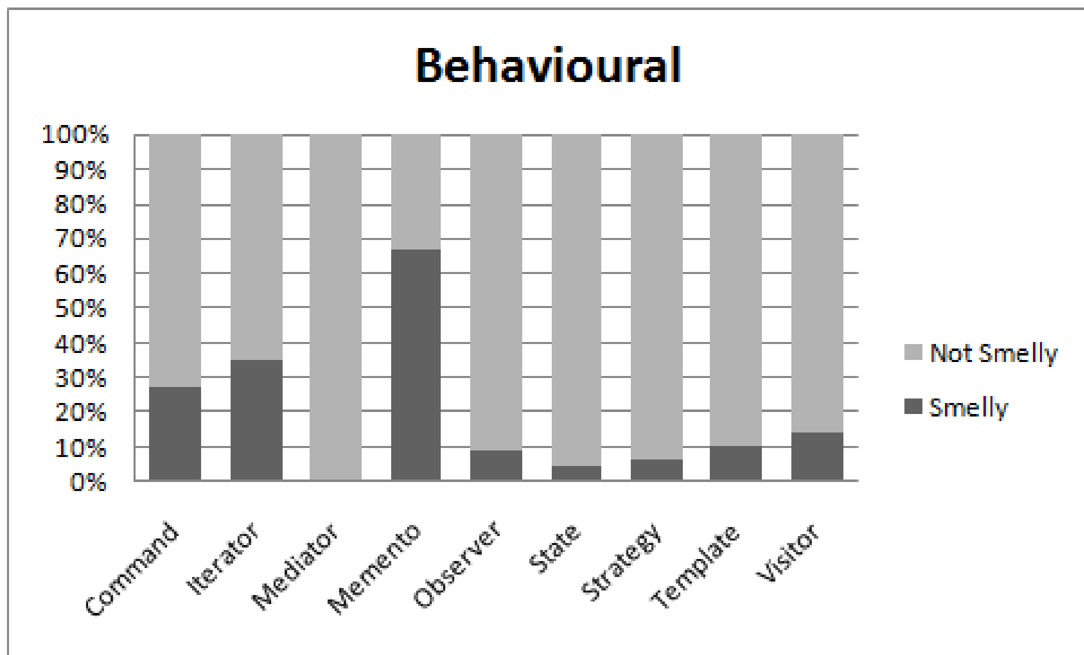


Figure 5.7: Comparison of smell proneness in behavioural category

might indicate that the subject design patterns in each category have different behaviours in the context of their links with smells. On the other hand, some

Table 5.7: P-values obtained from Kruskal Wallis test of each design pattern category

Category of Patterns	**p-value**
Creational	.048
Behavioural	.000
Structural	.007

design patterns have more instances than the others. For instance, in Figure 5.7, Iterator pattern appears to be more smell prone than Command pattern, although Command pattern has 58 instances of patterns while Iterator pattern has only 17 instances. In addition, the distribution of smells and design patterns instances is not equal. This observation might affect the conclusion of our results. Hence, a further analysis is needed. Next section provides more deep analysis by applying association rules to display possible significant relationships in the design pattern-code smell pairs.

5.1.3.1 Applying Association Rules for Design Patterns and Code Smells Pairs

As discussed in the Section 5.1.3, there is a need for further analysis to display relationships that might link specific design patterns with specific code smells. Consequently, "Association Rules" concept is employed in order to identify the relationships.

Table 5.8 presents data that matches each item from the studied design patterns with each item from the subject smells. The following acronyms are used for the columns captions: DC-Data Class, DCl-Data Clumps, RPB-Refused Parent Bequest Class, SC-Schizophrenic Class, BL-Blob, IC-Intensive Coupling, SD-

Table 5.8: Data of individual patterns with individual smells

DP \CS	DC	DCL	RPB	SC	BL	IC	SD	ID	ED	GC	FE	TB	MC	**SUM**[1]	**TOTAL**[2]
Abs. Factory	**2**	-	-	**1**	**1**	0	0	0	0	0	-	**1**	-	5	18
Builder	-	**1**	**1**	-	-	-	**2**	-	-	-	**1**	-	-	5	25
Factory Method	**2**	**1**	-	**3**	**4**	**1**	-	**1**	**2**	**1**	**1**	**1**	-	17	95
Prototype	-	-	-	-	-	-	-	-	-	-	-	-	-	-	21
Singlton	-	-	-	-	**1**	-	-	-	-	-	-	-	-	1	27
Adapter	-	-	-	**1**	**2**	-	-	-	-	-	-	**1**	-	4	58
Bridge	-	-	-	-	-	-	-	-	-	-	-	-	-	-	28
Composite	**2**	-	**1**	**2**	-	**1**	-	**1**	**3**	-	**1**	-	-	11	102
Facade	**1**	-	-	-	-	-	-	-	-	-	-	-	-	1	3
Decorator	-	-	-	-	-	-	-	-	-	-	-	-	-	-	57
Proxy	-	-	-	**1**	-	**1**	-	**1**	**2**	-	**1**	-	-	6	8
Command	-	-	-	-	**10**	**1**	-	-	**8**	**6**	-	-	**1**	26	58
Iterator	-	-	-	-	**3**	**1**	**2**	-	-	**2**	-	-	-	8	17
Mediator	-	-	-	-	-	-	-	-	-	-	-	-	-	-	3
Memento	**2**	-	-	-	**10**	-	-	-	**8**	**1**	-	-	-	21	18
Observer	**2**	-	-	**3**	**1**	-	**1**	-	-	-	-	-	-	7	87
State	-	-	-	-	-	-	-	-	-	-	**1**	**1**	-	2	66
Strategy	-	-	-	**1**	-	**1**	**2**	-	-	-	**1**	-	-	5	90
Template Method	**2**	-	**2**	**1**	**4**	**1**	-	**2**	**4**	**4**	**1**	**1**	-	22	133
Visitor	-	-	-	-	-	-	-	-	-	**1**	-	-	-	1	7

[1] *Pattern and smelly classes*
[2] *All classes which participate in DP.*

Sibling Duplication, ID-Internal Duplication, ED-External Duplication, GC-God Class, FE-Feature Envy, TB-Tradition Breaker and MC-Message Chains.

The data presented in Table 5.8 can lead directly to certain observations.

- Only 1 instance of a code smell was discovered to be contained in classes with the Singleton, Facade and the Visitor.

- Classes with the Prototype, Decorator and Bridge were discovered not to take place simultaneously with smells.

- The Blob, God Class and External Duplication smells are collocated with the Command patterns.

- The Blob and External Duplication are collocated with the Memento patterns.

- The Command and Memento patterns can take place simultaneously on a regular basis.

Mixtures of attributes in a data set can be expressed by Association Rules. The specific attributes are design patterns and code smells in the particular environment we observed. Two well-liked measures were utilized to classify the dependency rules between the attributes which were: *confidence* and *support* [118]. Weka tool has implemented some data mining techniques e.g Apriori algorithm [118].

In order to compute these measures, the assumption was made that, in each system, each class is a separate transaction. The following facts were subsequently established reagrding the transaction: (i) it includes an occurrence of a bad smell and (ii) it includes an occurrence of a design pattern. Each bad smell and each design pattern which is studied is refered to as an item set. Both metrics include values that range from (0-1), with higher values designating more important rules.

Support of an item set refers to the share of transactions which contain this itemset, thereby demonstrating its significance [118]. For example, if a system has 100 classes and 10 of these classes exhibit the bad smell Feature Envy, this can be taken to signify that, in this system, the Support of the Feature Envy is 10%. As a further example, the Support of the association of the Factory Method and God Class illustrates the proportion of transactions which include both the Factory Method and the God Class. Consequently, Support is shown to be a gauge of the frequency of an item in an association.

It is essential to be familiar with the naming conventions employed in the

association rules, which are antecedent and consequent, so that the concept of *Confidence* [118] can be properly understood. We regard design patterns to be the antecedent and bad smells to be the consequent for the purposes of this study. Confidence can be defined as the likelihood of observing the rules' consequent under the condition that the transactions include the antecedent. To put this another way, it represents the ratio between the Support of the association and the Support of the antecedent. The Confidence can be determined by using the below equation. The value of Confidence is usually higher if the consequent has a high level of Support. Due to this fact, it is more probable that the Support of the association is also high.

$$Conf(DP \rightarrow Smell) = Sup(DP \cup Smell) \div Sup(DP)$$

In this part of the evaluation, we decided to separately test the rule which combines (1)antecedent: design pattern on the left side and (2)consequent: the code smell on the right side. In order that even the weak rules could be documented for additional study, we set the minimal configuration values for support and confidence in Weka tool. The overall association rules which have been resultant are equal to (20 * 13 = 260). Only 5 rules are given in Table 5.9 with a low confidence ($< 95\%$). This was done because the majority of the rules in our data set exhibit very high confidence for the case which reflects the weak relation between both concepts: design pattern and code smell. A positive association between the presence of design patterns and code smells can be concluded from

Table 5.9: Extraction of filtered association rules

Rules
Command ⇒ *Blob*
Command ⇒ *GodClass*
Command ⇒ *ExternalDuplication*
Memento ⇒ *Blob*
Memento ⇒ *ExternalDuplication*

these results as shown in Table 5.9. A discussion and interpretation of the results are reported in the next section.

5.2 Discussion and Analysis

The findings reported in Section 5.1 are examined in this section. Explanations for the findings are given, and then related work is presented. After that, a review of the threats to the validity is discussed.

5.2.1 Co-occurrence between Design Patterns and Code Smells

In this section, the following observations can be made from Table 4.1 (page 46): the smelly classes are not widespread. In fact, it is estimated that about only 4.0% to 24.0% of classes in all systems are impacted by them. Furthermore, similarly only comprising 10.4% of all classes in the subject systems that have both attributes code smells and design patterns.

RQ1: Design class level

The relationship between the SnDP and the SDP for the systems, shown in Figure 5.2, changes over time. The observed relationship agrees with the work's final

conclusion (SDP $\leq$ SnDP) for the majority of systems. However, for the JUnit and the JHotDraw systems the situation is opposite. Data inspection of the systems exhibited that only 2 classes in the JUnit system that had a Schizophrenic smell were also represented in the Observer patterns. These relationships are not seen in other systems, a fact that implies that this was an isolated event due to suboptimal design choices. In JHotDraw, 5 associated classes including Blob, Schizophrenic, Data Clump, Refused Parent Bequest and Tradition Breaker, also took part in the Factory Method, Composite, Strategy and State. The bulk of these smelly classes were apparently associated with smells in only this project i.e. State with Tradition Breaker, Factory Method with Data Clumps and Strategy with Schizophrenic.

RQ2: Category level

The different categories of design patterns were analyzed with regard to the difference in smell proneness in order to identify if there is a significant variance between them. This result leads to answering RQ2. In Section 5.1.2, Tables 5.5 and 5.6 both a short and in depth summaries are given in order to identify the possible differences between design pattern categories in regards to smell proneness. The number of relevant cases varies from one pair to the other. Only 1 class of creational category is participating in the Nutch system, whilst Lexi system does not have any structural design patterns. It was noticed that every one of the different categories acts in the same way in terms of smell proneness. This observation tends towards the conclusion that the adoption of any of the design

patterns might produce the same reliable software. This is due to the fact that these design patterns usually exhibit similar percentages of smells.

Nevertheless, the behaviour category was found to have the highest absolute number of smells. The reason for this could be because of the fact that behavioural design patterns are relevant to the behaviour of the system, and consequently could be more prone to change. Furthermore, it is our belief that these design patterns represent difficult concepts. They are not readily comprehended and, as a result, they are not easy to apply to the design of software.

RQ3: Individual design patterns level

By examining the presence of individual smells in code, it is possible to discover the most common ones: Blob, God Class and External Duplication, which control other smells in the system. In the same way, the design patterns are not evenly distributed: the Template Method, Composite, Factory Method, Strategy, Observer and Command are the most employed instances in the subject systems. Such design patterns and code smells allotments comprise approximately 62% of the total number of smelly classes. Moreover, the Adapter with Decorator design patterns take part in an equal number of design pattern classes, although they are associated with a reduced number of smells, 4 and 0, respectively. In our search for a solution to answering RQ3, we centered on identifying the interested possible strong links between each design pattern and smell in the data set, characterized by "Association Rules".

Within a particular rule, we have a design pattern as an antecedent in the

left hand side, while a code smell plays the role of a consequent. Clearly, the majority of the important rules we discovered displays the mutual exclusive of design patterns and code smells. This sustains the findings resultant for RQ1. However, some rules represent a strong link between individual patterns with certain code smells, as revealed in Table 5.9. It can be seen from Table 5.8 that Singleton, State, Strategy, Adapter and Decorator are patterns where they are not allotted with smells generally, while it is opposite for Command and Memento patterns. As illustrated by Table 5.9, Command patterns are linked to the Blob, External Duplication and God Class smells, whereas the Memento patterns are connected with Blob and External Duplication, with the God Class being the exception. From an examination of our data set, the fact that Command and Memento co-exist in the classes is clear to see. Furthermore, explanations for our conclusion can be provided by the definitions and purposes of the Command and Memento patterns. Obviously, an excessive implementation of the patterns within the system evolution might potentially lead to the Blob and God Class in Command and Memento. Blob smells co-occur with External Duplication smells. Deep code analysis is required to clarify why they co-occur.

5.2.2 Comparison to Related Work

The literature provides little evidence regarding the part that the associations play in the interactions of smells and patterns which can be related to our findings. For this reason, it was decided to take an indirect approach to studying the interactions

between them. The method applied was a two-phase strategy, which examines the interactions between identified characteristics and qualities independent of the smells and patterns. The adverse effect of smells on maintainability has been reported in a number of cases, e.g. [89]. While the reverse assumptions have been made by other papers e.g. [119]. Smells continue to be regarded as a legitimate consideration for anticipating the effort put into maintenance. In tandem with smells, investigations have also been conducted into design patterns in regard to their effect on maintainability. Heged et al. [120] found that patterns make maintenance possible. In addition, Ng et al. [121] demonstrated that preceding understanding about patterns facilitates programmers to keep a software system running. Once combined, these relationships imply that the smells and patterns are correlated in some way negatively. Therefore, it should be anticipated that using patterns will give rise to a reduction in the reported cases of smelly classes. The rationale provided is sustained by the results of this study with exceptions in some cases. If defects are thought of as the main consideration which ties together smells and patterns, then the findings detailed in the literature are equivocal.

There are numerous studies where a correlation has been observed with the existence of smells and an increased defect ratio [88, 95, 122]. In contrast, a study by Voka [123] showed that certain patterns (Observer and Singleton) appear to display dissimilar actions. This contradicts our findings, which concluded that the Singleton was closely negatively correlated with the majority of the code smells. Research by Jaafar et al. [93] revealed that classes which depend on

smelly classes (known as anti-patterns) have a higher level of fault-proneness than others. Although this is not the case one hundred percent of the time for classes which exhibit dependencies on the classes which contribute to design patterns. It was also discovered by the authors that the use of patterns is commonly used by developers as a short-term cure for smells. Later both are eliminated from the system. While their findings have been achieved under conditions which differ slightly from the ones we experienced, where smells and patterns of the same class were used exclusively, they do at least partly sustain the conclusion that smells and patterns are typically mutually exclusive.

Research on the development of patterns and smells also suggests a range of conclusions. Aversano et al. [124] failed to show a major influence of individual patterns on change-proneness. However, the authors did come to the conclusion that design patterns appear to produce code which has a greater resistance to change. Moreover, they discovered that patterns are more suitable for applications which usually change more often. The results found by Bieman et al. [12] are inconclusive: in one case where systems were analyzed, the use of design patterns leads to a decrease in the change-proneness, while in four other systems the opposite effect was observed. In addition, the existence of code smells appears to raise the change proneness of the code. It was demonstrated by Olbrich et al. [88, 97] that there are certain smells which cause increased change-proneness. Other studies corroborated this result e.g. [122, 125] for further smells. It appears, however, that the development of code smells and design patterns is caused by different

stimuli. While the evolution of design patterns can be the result of the fact that the substitution of implementation classes inside a pattern can be achieved more easily. With the case of smells, the changes are meant to eliminate design issues.

Walter et al. [36] conducted an empirical study to evaluate the relationship between DPs and code smells. The authors have addressed the class level to study such relation. Their experiments are mainly done based on two open source projects: **jFreeChart and Apache Maven** with some subsequent releases of each project. The results indicate that the presence of design patterns is not strongly linked with the presence of code smell instances.

5.2.3 Threats to Validity

5.2.3.1 Construct Validity

Construct validity focuses on the measures used in the evaluation. The process of code smells detection is regarded as being especially important in this study. The various definitions of code smells are inherently ambiguous. The variations between the different smell detecting tools that are offered on the market add another layer of complexity. Accordingly, the findings can be significantly affected by the selection of a particular tool. In the case being studied, detection of both smells and patterns was achieved by using only a single detector. The fact that this was accomplished without cross-validation by another tool or with the input of human reviewers, could potentially give rise to false negative and false positive instances of the observed phenomena. The P-Mart repository has been produced

using different sources: studies in the literature [12]; Ptidej (pattern trace identification, detection, and enhancement in Java) tool for identifying design motifs [34]; and validation assignments for both undergraduate and graduate students. By using these various sources, it is possible to reduce the likelihood of false positive and negative instances of design patterns in the P-Mart repository.

5.2.3.2 Internal Validity

Internal validity can be defined as the extent to which the observed effects rely only on the intended experimental variables. The background of the developers can give rise to one source of threat to the internal validity. Whether the developers have been trained to work with DPs or not is not known. Nevertheless, the cause-and-effect relationship is not under investigation because it is not possible to control each variable that impacts the relationships between the different groups. The study is limited to attempting to determine if there is a significant connection between the targeted variables or not.

5.2.3.3 External Validity

External validity focuses on generalizations. The external validity of this study is endangered by the nature of the subject systems. Every subject system is open-source and has been created solely by using the Java programming language. To generalize the results obtained in this study, it is necessary to further explore the design patterns together with commercial systems and systems developed using other programming languages. To this end, this study should be regarded as an

initial step which it will be possible to reinforce at a later date with additional replications.

5.2.3.4 Conclusion Validity

Conclusion validity can be defined as the extent to which the conclusions that are made at the design level and the category level in ten different cases (i.e., ten subject systems in addition to when all of these systems are united). The distributed nature of the classes reduces the potential for bias in the classes in every system. In addition, most of the subject systems have a relatively close number of smelly classes. Conversely, two of the cases (Lexi and JUnit) have comparatively few classes and examples of design patterns. Different conclusions may be arrived at by taking into consideration a larger number of cases and systems derived from different criteria.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The relationship between design patterns and code (bad) smells has been explored within the scope of this study. In particular, co-existence between design patterns and bad smells were identified and investigated as shown in Chapter 5. The main steps followed in this study included improving our comprehension of the concepts related to it (Chapter 2). We conducted a literature review; completing an empirical study in which intriguing results were discovered; and then presenting an analysis and discussion of the results. The majority of the research uncovered in the literature review (Chapter 3) regarding the terms design patterns and bad smells highlight the refactoring opportunities. Conducting a literature review also revealed a number of incipient studies which studied and examined the co-

occurrences.

While surveying the studies in the literature (Chapter 3), most of them fit in three classifications: refactoring, structural, and co-occurrence. Many studies proposed tools that can provide algorithms which aim at detection of code smells and so apply automatic refactoring. Other studies have discussed the difference between both concepts, design pattern and code smells in terms of their structure. In particular, they used the structural characteristics of both design patterns and code smells to build a quality model. There are very few studies that performed an empirical study to analyze the co-occurrence relationship between design patterns and code smells. For instance, Seng et al. [103] have illustrated some examples where design patterns could lead to an emergence of code smells and design flaws. As an example, they used Facade design pattern to argue that Facade functions can only send requests to required classes which might affect the cohesion. Fontana et al. [106] showed cases where design patterns and code smells can co-occur. Strategy and Visitor design patterns are illustrated in their study to have an occurrence with Feature Envy smell. Therefore, identifying design patterns can be utilized in the code smells detection. The relation between design patterns and code smells is discussed in the field of software engineering. Yet, it is not deeply explored in the research. This study analyzes such relationship between design patterns and code smells and the possible situations where design patterns can co-occur with code smells.

We started performing the study (Chapter 4) by collecting the data of design

patterns and code smells. For the design patterns, P-mart repository was used instead of running a design pattern detection tool as the tools do not meet all the requirements (page 23). For the code smells detection, we ran InFusion tool over ten open source projects.

While conducting the empirical study (Chapter 5), the analysis of the study were separated into three levels: class level, category level, and individual patterns level. At the class level, all classes of the projects were considered in the analysis. We found that classes participating in design patterns display less smells than classes not participating in design patterns. Smell proneness and smell frequency were both considered in class level. They showed consistent results. Hence, we proceed to consider the smell proneness only in the other levels. Moreover, when doing the analysis on the category level, the result shows that categories almost have no significant differences in terms of smell proneness. Different powerful statistical techniques were used to conduct the analysis. However, at the level of individual design patterns, specific examples of design patterns connection with smells were discovered using association rules metrics, Support and Confidence. Although most of the rules showed weak relation between the presence of design patterns and the absence of code smells, it was observed that there is a connection that could potentially facilitate the production of bad smells. The results present the most noteworthy cases: the Command pattern with Blob and God Class smells. In addition, Memento pattern was discovered to be connected to Blob and External Duplication smells. On the other hand, Decorator pattern was

significantly not connected with smells.

The outcome of this thesis was to demonstrate the potential for the improper use of a design pattern, leading to the creation of code smells, whilst this may not hold true for certain cases. Consequently, this study has laid a foundation for future research and the potential for even more fascinating discoveries. It will be important for future studies to investigate additional concrete examples of design patterns that, because of their improper use, have classes that present bad smells. There is definitely the potential for future research in this area.

6.2 Future Work

Working on this thesis has given rise to numerous ideas, but due to time constraints, it was not possible to explore these in greater deep. The authors would like to recommend that further study be undertaken in the future as this would further enhance the comprehension of these programs. Additional extensions to this study can be presented below:

- The first potential area for future study is to reproduce the empirical study in the context of enterprise development, which could lead to additional data and consequently a higher degree of statistical significance. In addition to altering the target systems, there is potential to employ additional detection tools that could produce superior results and may also lead to the detection of additional or possibly different instances of bad smells. To ensure the success of this method, it is recommended to conduct it in a completely

controlled environment. This requires that the systems are familiar to and properly understood by the researchers. Under these conditions, the documentations of the employed design patterns must take place. In addition, another requirement could be that the system is not too big, which might also facilitate more accurate manual identification of bad smells. With the data in our possession, there is the potential to conduct an examination of the co-occurrences between design patterns and bad smells.

- As part of a future study it would be interesting to embark on a survey involving experienced professionals with the aim of analyzing code clips which include different design patterns and bad smells. This survey could potentially assess how these professionals react when they discover a co-occurrence of design patterns and bad smells.

- A potential work as a future work could be a smell prediction using metrics that connect to design patterns. The metrics can be modeled based on the availability of design patterns instances. In addition, smell detection can utilize clustering techniques in order to detect specific smells that connect to specific design patterns.

Bibliography

[1] M. Fowler, *Refactoring: improving the design of existing code.* Pearson Education India, 2009.

[2] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas. class corpus: A compiled version of the qualitas corpus," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–4, 2013.

[3] E. Gamma, *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[4] M. P. Cline, "The pros and cons of adopting and applying design patterns in the real world," *Communications of the ACM*, vol. 39, no. 10, pp. 47–49, 1996.

[5] L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *IEEE transactions on Software Engineering*, vol. 27, no. 12, pp. 1134–1144, 2001.

[6] K. Beck, R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick, and F. Paulisch, "Industrial experience with design patterns," in *Proceedings of the 18th international conference on Software engineering.* IEEE Computer Society, 1996, pp. 103–114.

[7] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-oriented software architecture, on patterns and pattern languages.* John wiley & sons, 2007, vol. 5.

[8] J. Kerievsky, *Refactoring to patterns.* Pearson Deutschland GmbH, 2005.

[9] R. Marinescu and M. Lanza, *Object-oriented metrics in practice.* Springer, 2006.

[10] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering,2009.* IEEE, 2009, pp. 75–84.

[11] P. Wendorff, "Assessment of design patterns during software reengineering: Lessons learned from a large commercial project," in *Fifth European Conference on Software Maintenance and Reengineering, 2001.* IEEE, 2001, pp. 77–84.

[12] J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander, "Design patterns and change proneness: An examination of five evolving systems," in *Proceedings Ninth international Conference on Software metrics symposium, 2003.* IEEE, 2003, pp. 40–49.

[13] M. Vokáč, W. Tichy, D. I. Sjøberg, E. Arisholm, and M. Aldrin, "A controlled experiment comparing the maintainability of programs designed with and without design patterns - a replication in a real programming environment," *Empirical Software Engineering*, vol. 9, no. 3, pp. 149–195, 2004.

[14] C. Jebelean, C.-B. Chirilă, and V. Creţu, "A logic based approach to locate composite refactoring opportunities in object-oriented code," in *IEEE International Conference on Automation Quality and Testing Robotics (AQTR),2010*, vol. 3. IEEE, 2010, pp. 1–6.

[15] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonca, "Identifying code smells with multiple concern views," in *Brazilian Symposium on Software Engineering (SBES)*. IEEE, 2010, pp. 128–137.

[16] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[17] C. Bouhours, H. Leblanc, and C. Percebois, "Sharing bad practices in design to improve the use of patterns," in *Proceedings of the 17th Conference on Pattern Languages of Programs*. ACM, 2010, p. 22.

[18] F. Khomh, "Patterns and quality of object-oriented software systems," Ph.D. dissertation, 2010.

[19] W. B. McNatt and J. M. Bieman, "Coupling of design patterns: Common practices and their benefits," in *COMPSAC 25th Annual International Computer Software and Applications Conference, 2001.* IEEE, 2001, pp. 574–579.

[20] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Analysing anti-patterns static relationships with design patterns," *Proc. PPAP*, vol. 2, p. 26, 2013.

[21] G. Vale, E. Figueiredo, R. Abílio, and H. Costa, "Bad smells in software product lines: A systematic review," in *Eighth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS).* IEEE, 2014, pp. 84–94.

[22] B. F. Webster, "Pitfalls of object oriented development," 1995.

[23] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.

[24] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th international symposium on Software visualization.* ACM, 2010, pp. 5–14.

[25] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: research and practice*, vol. 23, no. 3, pp. 179–202, 2011.

[26] J. A. de Oliveira, E. M. Fernandes, and E. Figueiredo, "Evaluation of duplicated code detection tools in cross-project context," *3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pp. 49–56, 2015.

[27] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 97–106.

[28] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[29] J. Dong, D. S. Lad, and Y. Zhao, "Dp-miner: Design pattern discovery using matrix," in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems ECBS'07.* IEEE, 2007, pp. 371–380.

[30] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[31] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the 24th international conference on Software engineering.* ACM, 2002, pp. 338–348.

[32] F. A. Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Information sciences*, vol. 181, no. 7, pp. 1306–1324, 2011.

[33] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code," in *21st IEEE/ACM International Conference on Automated Software Engineering ASE'06*. IEEE, 2006, pp. 123–134.

[34] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag*, 2005.

[35] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, 2006.

[36] B. Walter and T. Alkhaeir, "The relationship between design patterns and code smells: An exploratory study," *Information and Software Technology*, vol. 74, pp. 127–142, 2016.

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

[38] J. Dietrich and C. Elgar, "Towards a web of patterns," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 108–116, 2007.

[39] A. Binun and G. Kniesel, "Dpjf-design pattern detection with high accuracy," in *2012 16th European Conference on Software Maintenance and Reengineering (CSMR).* IEEE, 2012, pp. 245–254.

[40] F. A. Fontana, S. Maggioni, and C. Raibulet, "Understanding the relevance of micro-structures for design patterns detection," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2334–2347, 2011.

[41] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Behavioral pattern identification through visual language parsing and code instrumentation," in *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR'09.* IEEE, 2009, pp. 99–108.

[42] A. De lucia, V. Deufemia, C. Gravino, and M. Risi, "Improving behavioral design pattern detection through model checking," in *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010.* IEEE, 2010, pp. 176–185.

[43] M. L. Bernardi, M. Cimitile, and G. A. Di Lucca, "A model-driven graph-matching approach for design pattern detection," in *20th Working Conference on Reverse Engineering*, 2013, pp. 172–181.

[44] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A cooperative parallel search-based software engineering approach for code-smells detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[45] O. Ciupke, “Automatic detection of design problems in object-oriented reengineering,” in *30 Proceedings of Technology of Object-Oriented Languages and Systems*. IEEE, 1999, pp. 18–32.

[46] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, “Detecting defects in object-oriented designs: using reading techniques to increase software quality,” in *ACM Sigplan Notices*, vol. 34, no. 10. ACM, 1999, pp. 47–56.

[47] G. Ganea, I. Verebi, and R. Marinescu, “Cassessment with incode,” *Science of Computer Programming*, 2015.

[48] J. Dexun, M. Peijun, S. Xiaohong, and W. Tiantian, “Detection and refactoring of bad smell caused by large scale,” *International Journal of Software Engineering & Applications*, vol. 4, no. 5, p. 1, 2013.

[49] A. Trifu and R. Marinescu, “Diagnosing design problems in object oriented systems,” in *12th Working Conference on Reverse Engineering*. IEEE, 2005, pp. 10–pp.

[50] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, “Domain-specific tailoring of code smells: an empirical study,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 167–170.

[51] M. Salehie, S. Li, and L. Tahvildari, “A metric-based heuristic framework to detect object-oriented design flaws,” in *14th IEEE International Conference on Program Comprehension, 2006. ICPC 2006.* IEEE, 2006, pp. 159–168.

[52] R. Marinescu, G. Ganea, and I. Verebi, “incode: Cassessment and improvement,” in *14th European Conference on Software Maintenance and Reengineering (CSMR), 2010.* IEEE, 2010, pp. 274–275.

[53] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, “Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems,” in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development.* ACM, 2012, pp. 167–178.

[54] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, “Supporting the identification of architecturally-relevant code anomalies,” in *28th IEEE International Conference on Software Maintenance (ICSM), 2012.* IEEE, 2012, pp. 662–665.

[55] I. H. Moghadam and M. O. Cinneide, “Automated refactoring using design differencing,” in *16th European conference on Software maintenance and reengineering (CSMR), 2012.* IEEE, 2012, pp. 43–52.

[56] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó. Cinnéide, and K. Deb, “On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach,” *Empirical Software Engineering*, vol. 21, no. 6, pp. 2503–2545, 2016.

[57] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “Detecting bad smells in source code using change history informa-

tion," in *2013 IEEE/ACM 28th international conference on Automated software engineering (ASE).* IEEE, 2013, pp. 268–278.

[58] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in *IEEE International Conference on Software Maintenance.* IEEE, 2007, pp. 519–520.

[59] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[60] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *29th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2013, pp. 392–395.

[61] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[62] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 1, p. 6, 2014.

[63] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[64] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[65] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3-4, pp. 345–361, 2010.

[66] T.-W. Kim, T.-G. Kim, and J.-H. Seu, "Specification and automated detection of code smells using ocl," *International Journal of Software Engineering and Its Applications*, vol. 7, no. 4, pp. 35–44, 2013.

[67] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.* IEEE, 2003, pp. 91–100.

[68] M. J. Munro, "Product metrics for automatic identification of" bad smell" design problems in java source-code," in *Software Metrics, 2005. 11th IEEE International Symposium.* IEEE, 2005, pp. 15–15.

[69] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Fifth European Conference on Software Maintenance and Reengineering.* IEEE, 2001, pp. 30–38.

[70] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonca, "Identifying code smells with multiple concern

views," in *2010 Brazilian Symposium on Software Engineering (SBES),*. IEEE, 2010, pp. 128–137.

[71] A. A. Rao and K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix 1," 2007.

[72] M. Nitin, "Java smell detector," Ph.D. dissertation, Master Thesis, San Jose State University, 2011.

[73] W. Abdelmoez, E. Kosba, and A. F. Iesa, "Risk-based code smells detection tool," in *The International Conference on Computing Technology and Information Management (ICCTIM)*. Society of Digital Information and Wireless Communication, 2014, p. 148.

[74] N. Roperia, "Jsmell: A bad smell detection tool for java systems," Ph.D. dissertation, 2009.

[75] A. Yamashita and L. Moonen, "To what extent can maintenance problems be predicted by code smell detection?–an empirical study," *Information and Software Technology*, vol. 55, no. 12, pp. 2223–2242, 2013.

[76] K. Nongpong, "Integrating" code smells" detection with refactoring tool support," 2012.

[77] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 116–125.

[78] A. Rani and H. Kaur, "Detection of bad smells in source code according to their object oriented metrics," *International Journal for Technological Research in Engineering*, vol. 1, no. 10, pp. 1211–14, 2014.

[79] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE, 2002, pp. 97–106.

[80] S. Slinger, "Code smell detection in eclipse," *Delft University of Technology*, 2005.

[81] P. Danphitsanuphan and T. Suwantada, "Code smell detecting tool and code smell-structure bug relationship," in *Spring Congress on Engineering and Technology (S-CET).* IEEE, 2012, pp. 1–5.

[82] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *29th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2013, pp. 396–399.

[83] M. Gatrell, S. Counsell, and T. Hall, "Design patterns and change proneness: a replication using proprietary c# software," in *16th Working Conference on Reverse Engineering, 2009. WCRE'09.* IEEE, 2009, pp. 160–164.

[84] M. Gatrell and S. Counsell, "Design patterns and fault-proneness a study of commercial c# software," in *Fifth International Conference on Research Challenges in Information Science (RCIS), 2011.* IEEE, 2011, pp. 1–8.

[85] M. O. Elish and M. A. Mohammed, "Quantitative analysis of fault density in design patterns: An empirical study," *Information and Software Technology*, vol. 66, pp. 58–72, 2015.

[86] J. Rudzki, "How design patterns affect application performance–a case of a multi-tier j2ee application," in *International Workshop on Scientific Engineering of Distributed Java Applications.* Springer, 2004, pp. 12–23.

[87] J. L. Krein, L. J. Pratt, A. B. Swenson, A. C. MacLean, C. D. Knutson, and D. L. Eggett, "Design patterns in software maintenance: An experiment replication at brigham young university," in *Second International Workshop on Replication in Empirical Software Engineering Research (RESER), 2011*. IEEE, 2011, pp. 25–34.

[88] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2010, pp. 1–10.

[89] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *28th IEEE International Conference on Software Maintenance (ICSM).* IEEE, 2012, pp. 306–315.

[90] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[91] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[92] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *10th International Conference on Quality Software, 2010*. IEEE, 2010, pp. 23–31.

[93] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults," *Empirical Software Engineering*, vol. 21, no. 3, pp. 896–931, 2016.

[94] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 75–84.

[95] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of systems and software*, vol. 80, no. 7, pp. 1120–1128, 2007.

[96] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 18.

[97] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 3rd international symposium on empirical software engineering and measurement.* IEEE Computer Society, 2009, pp. 390–400.

[98] E. Arisholm and D. I. Sjoberg, "Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software," *IEEE Transactions on software engineering*, vol. 30, no. 8, pp. 521–534, 2004.

[99] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European conference on Software maintenance and reengineering (CSMR).* IEEE, 2011, pp. 181–190.

[100] F. Khomh, "Squad: Software quality understanding through the analysis of design," in *16th Working Conference on Reverse Engineering, 2009.* IEEE, 2009, pp. 303–306.

[101] M. von Detten and S. Becker, "Combining clustering and pattern detection for the reengineering of component-based software systems," in *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS.* ACM, 2011, pp. 23–32.

[102] I. Polasek, P. Liska, J. Kelemen, and J. Lang, "On extended similarity scoring and bit-vector algorithms for design smell detection," in *IEEE 16th International Conference on Intelligent Engineering Systems (INES), 2012*, 2012.

[103] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, pp. 1909–1916.

[104] A. Trifu, *Towards Automated Restructuring of Object Oriented Systems*. Univ.-Verlag Karlsruhe, 2008.

[105] J. Pérez and Y. Crespo, "Perspectives on automated correction of bad smells," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM, 2009, pp. 99–108.

[106] F. A. Fontana and S. Spinelli, "Impact of refactoring on quality code evaluation," in *Proceedings of the 4th Workshop on Refactoring Tools*. ACM, 2011, pp. 37–40.

[107] C. Dorman *et al.*, "Software change in the solo iterative process: An experience report," in *Agile Conference (AGILE), 2012*. IEEE, 2012, pp. 21–30.

[108] M. Alshayeb, “The impact of refactoring to patterns on software quality attributes,” vol. 36, no. 7. Springer, 2011, pp. 1241–1251.

[109] M. ALshayeb, “Empirical investigation of refactoring effect on software quality,” vol. 51, no. 9. Elsevier, 2009, pp. 1319–1326.

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

[111] “InFusion,” http://www.intooitus.com/inFusion.html, [Accessed on 12-1-2014].

[112] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel, “iplasma: An integrated platform for quality assessment of object-oriented design,” in *In ICSM (Industrial and Tool Volume)*. Citeseer, 2005.

[113] F. Wilcoxon, “Individual comparisons by ranking methods,” *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[114] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures.* crc Press, 2003.

[115] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *The annals of mathematical statistics*, pp. 50–60, 1947.

[116] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American statistical Association*, vol. 47, no. 260, pp. 583–621, 1952.

[117] H. Levene *et al.*, "Robust tests for equality of variances," *Contributions to probability and statistics*, vol. 1, pp. 278–292, 1960.

[118] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Acm sigmod record*, vol. 22, no. 2. ACM, 1993, pp. 207–216.

[119] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2639–2653, 2013.

[120] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, "Myth or reality? analyzing the effect of design patterns on software maintainability," in *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity.* Springer, 2012, pp. 138–145.

[121] T. H. Ng, Y. T. Yu, S. C. Cheung, and W. K. Chan, "Human and program factors affecting the maintenance of programs with deployed design patterns," *Information and Software Technology*, vol. 54, no. 1, pp. 99–118, 2012.

[122] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 17–23.

[123] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 904–917, 2004.

[124] F. Arcelli, S. Masiero, and C. Raibulet, "Elemental design patterns recognition in java," in *13th IEEE International Workshop on Software Technology and Engineering Practice*. IEEE, 2005, pp. 196–205.

[125] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Seventh International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

Vitae

Personal details:

- Name: Mahmoud Abdulkarim Alfadel
- Nationality: Syrian
- Date of Birth: 05/01/1991
- Personal Email: *alfadelmahmood@gmail.com*
- Permanent Address: Saudi Arabia - Qassim

Education, Research, and Experience:

- BS degree in Information Technology and Software Engineering from Damascus University, Syria (September 2008 - Septemeber 2013).
- Research Interests: Empirical software engineering, software metrics, design patterns, secure software.
- Academic Experience: Research Assistant, KFUPM University, Saudi Arabia (January 2015 - May 2017).
- Mastered Programming Languages: C, Java, HTML, JavaScript, ASP.NET, PHP, SQL.