# SOFTWARE BAD SMELLS PRIORITIZATION MODEL

BY
Turki Rujaian Alshammari

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

January 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

**DEANSHIP OF GRADUATE STUDIES**

This thesis, written by **Turki Rujaian Alshammari** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING.**

Alshayb 7/1/2018

Dr. Mohammad Alshayeb
(Advisor)

Sajjad Mahmood 07/01/2018

Dr. Sajjad Mahmood
(Member)

Dr. Khalid Aljasser
Department Chairman

Jan 8, 2018

Dr. Khalid Aljasser
(Member)

Dr. Salam A. Zummo
Dean of Graduate Studies

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

9/1/18

Date

*Dedicated*

*to*

*my lovely wife*

# ACKNOWLEDGMENTS

I am thankful to Allah for who gave me the health and courage to finish this work.

I would like to express my deep appreciation to Dr. Mohammad Alshayeb who my thesis advisor for his help, patience, encouragement and extraordinary support. Words cannot express how grateful and appreciative I am of his help to finish this research.

In addition, I would like to thank my committee members Dr. Sajjad Mahmood and Dr. Khalid Aljasser for their feedback and support during this thesis work. I want to thank King Abdulaziz City for Science and Technology (KACST) for providing financial support for this research under grant number أ ط-37-1043.

I also would like to thank my parent, my family, and friends for their love and support throughout my study.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ABSTRACT

Full Name : Turki Rujaian Alshammari

Thesis Title : Software Bad Smells Prioritization Model

Major Field : Software Engineering

Date of Degree : May, 2017

Bad smell are parts of the codes that might have a problem. They are indicators to possible problems in the software. Whether they are trivial or complex, bad smells are shown to be of great impact on the software quality. To improve the maintainability of the software, and to mitigate and resolve bad smells, refactoring methods are used. Refactoring is the process of changing the internal behavior of the system without affecting its external behavior. Although the study of bad smell impact on software quality does exist, maintainers find it hard to refactor all the bad smells in the software because of constraints like time and cost. Hence, research in the area of bad smell prioritizations and ranking is needed. The aim of this research is to present a model to prioritize bad smells based on their impact on software maintainability. In this research, a prioritization model is proposed to rank bad smells based on their impact on software maintainability. The model is validated against five bad smells and five three open-source projects. The validation is performed by comparing the maintainability value before and after the removal of bad smells. In addition, a visualization of the relationships between classes' maintainability and its bad smells ranking is presented. We conclude that bad smells have different impact on software maintainability. Some classes showed a significant improvement while others show a decrease in quality.

# ملخص الرسالة

**الاسم الكامل**: تركي رجيعان الشمري

**عنوان الرسالة:** نموذج ترتيب الروائح الكريهة للبرمجيات

**التخصص:** هندسة البرمجيات

**تاريخ الدرجة العلمية**: مايو 2017

المؤشرات السلبية للبرمجيات هي أجزاء من الكود البرمجي والتي قد تحوي على مشاكل، فهي مؤشرات لمشاكل محتملة في المنتج البرمجي. ومهما كانت المؤشرات السلبية للبرمجيات بسيطة أو معقدة، إلا أن أثرها على جودة المنتج البرمجي واضح. لتحسين عملية صيانة المنتج البرمجي ولتجنب وحل المؤشرات السلبية في البرمجيات، يتم استخدام أساليب إعادة الهيكلة. إعادة الهيكلية هي عملية تحسين البرامج عن طريق تغيير هيكلها الداخلي بدون التأثير على السلوك الخارجي. وبالرغم من وجود دراسات عديدة في أثر المؤشرات السلبية للبرمجيات على جودة المنتج البرمجي، إلا ان الأشخاص القائمين على صيانة المنتج البرمجي يجدون صعوبة في إعادة هيكلة المنتج للتخلص من كل المؤشرات السلبية بسبب عوامل عديدة مثل الوقت والتكلفة. لذلك فإن هناك حاجة للبحث في مجال ترتيب المؤشرات السلبية للبرمجيات.

الهدف من هذا البحث هو عرض نموذج لترتيب المؤشرات السلبية للبرمجيات بناء على تأثيرها على صيانة المنتج البرمجي. في هذا البحث، تم اقتراح نموذج لترتيب المؤشرات السلبية بناء على تأثيرها على صيانة المنتج البرمجي، وتم التحقق من صحته باستخدام خمسة أنواع من المؤشرات السلبية للبرمجيات وخمسة منتجات برمجية مفتوحة المصدر. تمت تنفيذ عملية التحقق وذلك بمقارنة سهولة صيانة المنتج البرمجي قبل وبعد إزالة المؤشرات السلبية للبرمجيات. بالإضافة إلى ذلك، تم تصوير العلاقة بين صيانة المنتج البرمجي وترتيب المؤشرات السلبية الموجودة فيه. نستنتج من هذا البحث أن المؤشرات السلبية للبرمجيات لها تأثير مختلف على صيانة المنتج البرمجي، فبعضها أظهر تحسناً في قيمة الصيانة فيما أظهر البعض الآخر ترديا في القيمة.

# CHAPTER 1

# INTRODUCTION

Software maintainability is considered to be of most important when it comes to software quality. It is viewed as a fundamental attribute of the system quality. Software maintainability, informally defined as the degree of ease a to alter and modify a system, can be improved by performing maintenance tasks or activities, or best known in the software development life-cycle as the maintenance process.

The aim of the maintenance process is to improve the quality of the software. The quality improvement can be achieved by applying certain maintainability tasks. Those tasks vary in terms of their end goal. Some tasks aim to correct errors and faults reported on the software. We call these tasks corrective tasks. Other tasks goal is to improve the functionalities of the software, by adding functionalities, removing functionalities, or modifying existing ones. We call this type of tasks perfective tasks. Other tasks, however, focuses on improving the quality of the software code to improve its performance. Those tasks are known as performance tasks.

One of the main challenges software managers face is the ability to keep up with the rate the software is changing. As the software evolve, its structure keeps evolving and changing and the complexity of the software keeps increasing. This will make the process of maintaining the software harder and harder with each release. This is because that the software increases in terms of size and functionality. This is bound to happen for many reasons like: the software need to cope up with competitors, or the customer requirement has been changed. Hence, as a result if that, the

ability to understand the system and analyze it will decrease. This means that software managers and maintainers need to identify the parts of the software that was affected by the evolution process. As the software evolves, you may find that parts of its code are not well-written or optimized. These parts need to be given attention as early as possible. If they are not fixed early on, they may have ripple effect on other parts of the software. Also, they may slow the maintenance process and increase the cost dramatically if they were to be left in the software without fixing. To give those parts a name, the term Bad Smell was coined [1].

Bad smells are seen as an indication to a serious problem in the software. One of the main characteristics is the fact that they are easy to notice in most cases. For example, you can easily spot a large method in your class by seeing the lines of code. This is called Large Method bad smell. Also, you can easily spot the large classes (Large Class bad smell) in your software by analyzing the lines of code for each class. Another characteristic of bad smells is that they don't indicate a problem in the software in all the cases. The existence of Large Class might be acceptable in some cases. You need to investigate each case to see if it really reflects a problem in the underlying software.

## 1.1 Problem Description

Although the focus on the impact of bad smells on software quality have been insightful in the process of understanding bad smells and improving the software quality, the impact of one bad smell compared to another has not been explored. It will be helpful to know which bad smell will affect quality more.

Knowing which bad smell has greater impact than another will help software professionals focus on the more important smells. Namely, some of the key challenges are:

- **Maintainability Cost:** As the software grows bigger and bigger, maintainer need to face the fact that maintaining and improving the whole system might not be possible. It is essential that they have the knowledge and tools to do as little maintenance tasks as possible while getting the highest quality improvement possible.

- **Lack of bad smell model:** In order to favor refactoring one bad smell over the other, there needs to be a model that evaluates and prioritize them. One of the main reasons behind this research is to investigate the relationships between bad smells, as opposed to investigate the effect of one bad smell on the quality. The literature is focusing on enhancing methods and tools to detect code bad smells [2] whereas the improvement of our understanding to these bad smells is not investigated enough.

## 1.2 Motivation

The advantages of constructing such prioritization model are numerous to help both researchers and professionals in understanding the extent of the effect those bad smells could have on the software. Here are some of the advantages:

a) **Reduce Cost:** one of the main advantages of such model is reducing cost. This will be possible by knowing which bad smells are more impactful on software quality than others. Designs can then remedy these severe smells by applying refactoring.

b) **Improve Quality:** this model will help in improving the overall quality of the software. Having such model will assist in deciding the priorities in addressing bad smells.

c) **Shifting Focus:** with this model, industry professionals will have a way to spend more effort on studying, analyzing, and refactoring a subset of bad smells that have more effect on software quality.

d) **Encouraging Further Research:** There is a lack of research in the literature governing the area of prioritizing bad smell effect. A prioritization model can assist academic researchers in exploring bad smells furthermore by pointing the types of bad smells that need to be addresses more rigorously.

## 1.3 Research Objective

Knowing the impact on software quality of one bad smell compared to other bad smells will help enhance the overall quality of said software. The goal of this research is to identify the impact of bad smells on software quality, and then construct a model that will help in ranking bad smells according to their impact. Our work addresses the following research questions:

- What are the effects of bad smells on software quality, if any?
- Which bad smells will improve the software quality more than other?
- How bad smells are related to each other in terms of their effect on quality?
- Specifically, the objectives of this research are:
- To identify the impact of bad smells on software maintainability.
- To provide a ranking model for bad smells based on their impact.
- To visualize the impact and relationships of bad smells

## 1.4 Research Contribution

This research will contribute to the academic and industrial communities in the following ways:

1. A Classification of Bad Smells based on their effect on different software maintainability.
2. A Prioritization Model of Bad Smells to rank different bad smells based on how much they affect software maintainability. This will help the industrial professionals to distribute their effort more efficiently.
3. A visualization of the impact of bad smells on software maintainability and how the impact of one bad smell will affect the others.

## 1.5 Outline of Thesis

The rest of this thesis is divided as follows:

- Chapter 2 presents some background information about the work this research is based on. The chapter describes the concept of code bad smells and code bad smell detection. Next, the chapter presents the concept of refactoring bad smells. Finally, the chapter describes the idea of maturity models for software.
- Chapter 3 discusses the surveys the literature in the fields related to our research. The first section presents the research done in the effect of bad smell on quality. The second section summarized some of the maturity and prioritization model in the literature.
- Chapter 4 describes the research methodology used in this research along with a proposal for a modified maturity model to measure maintainability. The main components of the

research methodology are described in depth. In addition to that, the maturity model used is explained in detail.

- Chapter 5 describes the empirical evaluation for software maintainability impact. The first sections outline the experiment design: context selection, hypothesis formulation, variables selection, and subject systems. The second part of the chapter presents the results of the experiment and evaluate the hypothesis. The last part presents the impact of bad smell on maintainability.

- Chapter 6 presents our proposed bad smells prioritization mode. The first part defines the proposed approach. The second part applied the approach to the result of our experiment in the previous chapter. The third part presents expert evaluation of our proposed model. The last section explores the relationship between the ranking of bad smells and their impact on maintainability.

- Chapter 7 concludes our research by outlining the result and contribution of the research. It also mentions threats to validity and future work.

# CHAPTER 2

# BACKGROUND

This chapter describes some background information about the main concepts discussed in this research. Namely, the following section will provide some explanations about maturity models, code smell detection and refactoring.

## 2.1 Refactoring

Software code is bound to evolve and change with time. Features are added, changed, and removed over time. This constant change and evolvement will often lead to a complex hard to maintain system. To solve this issue, refactoring methods were introduced [1]. Refactoring methods are techniques to help in restructuring the internal code without affecting the external functionalities of the system.

The result of a refactoring activity is an easier to maintain software that does the same functionalities of the original one. It is an important part of the software maintenance and development life-cycle, especially when it is known that most of the cost of the software development life-cycle is consumed by the maintenance process [3]. Refactoring helps in improving the understandability of the software, and simplify the process of modifying and maintaining the software.

To demonstrate the process of refactoring, we will explain how one refactoring methods is carried out: Extract Class. Assume you have a class called Person; the person has a name and an office

number, with a method to get the phone number. This class represents two different entities, Person and Telephone Number. It makes sense to divide this class to two classes, where each class contains its closely related data. Then you can have one class using the other class. Therefore, to perform the refactoring method Extract Class, we do the following: (1) create a new class called TelephoneNumber, (2) move attributes related to the Telephone Number to the new class, (3) set up accessor method to get the telephone number, (4) modify the Person class so that it now uses Telephone Number class. This refactoring method is visualized in Figure 1

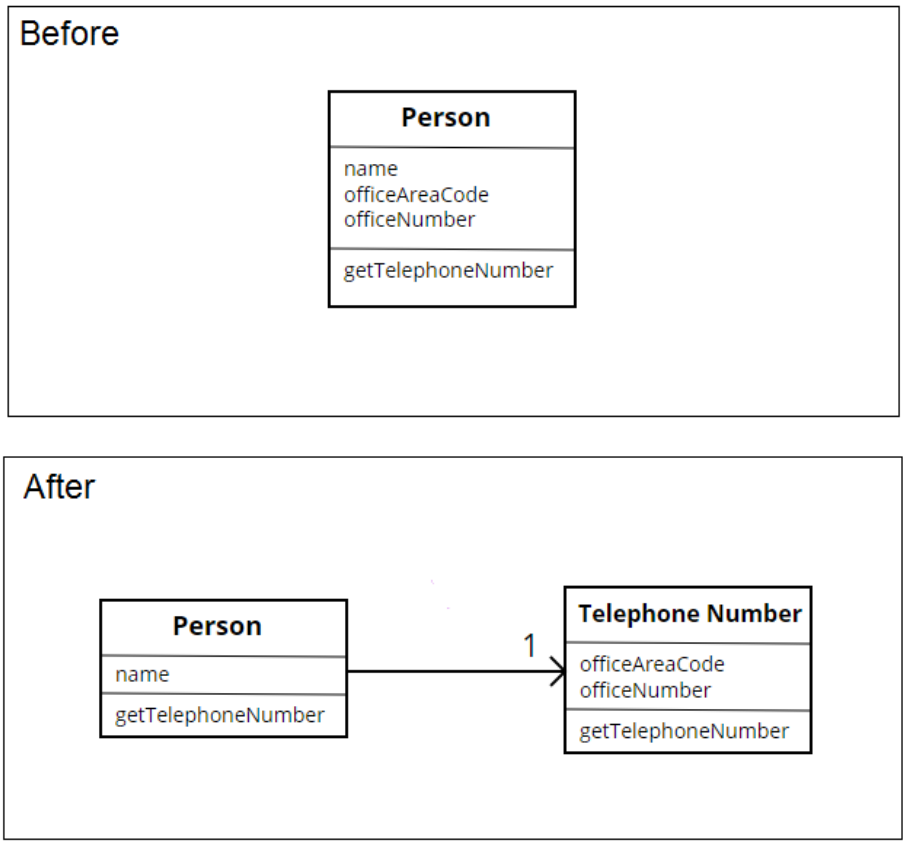



**Figure 1: An Example of Extract Class refactoring**

## 2.2 Bad Smells

Code Bad Smells are described as part of the code that needs more attention. They are technically not an error in the software but a problem in the code that needs to be addressed. These code smells range from a straightforward copying and pasting of code (Duplicate Code) to a complicated problem in the software structure (Shotgun Surgery). Fowler et al. [1] defined 22 code smells and described the needed steps to get rid of them; i.e. refactor them in Table 1. These smells can lead to decay in certain aspects of the software quality; like analyzability, correctness, reusability, etc. These code smells are commonly used to detect refactoring possibilities in software [4].

**Table 1: Bad Smells and Possible Refactoring Methods.**

| No. | Bad Smell Name | Possible Refactoring Methods |
|---|---|---|
| 1 | Duplicated Code | Extract Method, Pull Up Method, Substitute Algorithm. |
| 2 | Long Method | Extract Method, Replace Temp With Query, Introduce Parameter Object and Preserve Whole Object. Replace Method with Method Objects, Decompositional Objects |
| 3 | Large Class | Extract Class, Extract Sub Class. Duplicate Observed Data |
| 4 | Long Parameter List | Replace Parameter with Method |
| 5 | Divergent Change | Extract Class |
| 6 | Shotgun Surgery | Move Class or Move Method |
| 7 | Feature Envy | Move Method and Extract Method |
| 8 | Data Clumps | Extract Class, Introduce Parameter Object and Preserve Whole Object |
| 9 | Primitive Obsession | Replace Data Value With Object, Replace Type Code With Class, Replace Type Code With Subclass., Replace Type Code With State Strategy, Extract Class, Introduce Parameter Object And Replace Array With Object |
| 10 | Switch Statements | Extract Method, Move Method, Replace Type Code with Subclass or Replace Type Code with State |
| 11 | Parallel Inheritance Hierarchies | Move Method or Move Field |
| 12 | Lazy Class | Collapse Hierarchy or Inline Class |
| 13 | Speculative Generality | Collapse Hierarchy, Inline Class, Remove Parameters Methods, Remove Methods |
| 14 | Temporary Fields | Extract Class, Introduce Null Objects |
| 15 | Message Chains | Hide Delegate, Extract Method and Move Method. |

| | | |
|---|---|---|
| 16 | Middle Man | Remove Middle Man, Inline Method, Replace Delegation with Inheritance |
| 17 | Inappropriate Intimacy | Move Method or Move Field, Change Bidirectional Association to Unidirectional, Extract Class, Hide Class |
| 18 | Alternative Classes with different interfaces | Rename Method, Move Method, Extract Superclass |
| 19 | Incomplete Library Class | Move Method, Introduce Foreign Method, Introduce Local Extension |
| 20 | Data Class | Encapsulate Field or Encapsulate Collection, Remove Setting Method, Move Method or Extract Method, Hide Method |
| 21 | Refused Bequest | Push Down Method Push Down Field, Replace Inheritance with Delegation |
| 22 | Comments | Extract Method or Rename Method, Introduce Assertion. |

Mantyla [5] suggested a bad smells taxonomy where he grouped bad smells based on the common concepts they share with each other. All the 22 bad smells are divided into seven categories which are presented in Table 2. They are as follows:

a) The Bloaters: Classes and methods that have been growing larger and larger to the point where managing them is not effective.

b) The Object-Orientation Abusers: The common thing between these types of smells is that they violate the object-oriented design, like inheritance misuse or violating the information hiding principle.

c) The Change Preventers: Change Preventer smells thwart the changing of software. Classes with these smells tend to be difficult to modify.

d) The Dispensables: The code smells in this category share the fact that they all represent something redundant that should be taken out from the software.

e) The Encapsulators: This group contains two code smells dealing with encapsulation, where decreasing one smell will result in an increase in the other.

f) The Couplers: The code smells in this group represent high coupling which violates the object-oriented design principle of minimal coupling.

g) Others: This group has two code smells that do not fit in the other groups.

**Table 2: Taxonomy of Code Bad Smells**

| No. | Categories | Bad Smells |
|---|---|---|
| 1 | Bloaters | Long Method, Large Class, Data Clumps, Primitive Obsession and Long Parameter List |
| 2 | Object Orientation Abusers | Switch Statements, Temporary Field, Refused Bequest and Alternative Classes with Different Interfaces |
| 3 | Change Preventers | Divergent Change, Shotgun Surgery and Parallel Inheritance Hierarchies |
| 4 | Dispensables | Lazy class, Data class, Duplicate Code, Dead Code and Speculative Generality |
| 5 | Encapsulators | Message Chains and Middle Man |
| 6 | Couplers | Feature Envy and Inappropriate Intimacy |
| 7 | Others | Incomplete Library Class and Comments |

## 2.3 Bad Smells Detection

According to a recent review of the current trends of bad smells research, nearly half of the research focuses on tools and methods to detect bad smells in software [2]. Numerous techniques and tools have been developed to detect and identify bad smells. Most commonly, software metrics are used to identify bad smells in code [6, 7].

For example, Lanza and Marinescu [8] suggested that we can detect when a method has a Feature Envy smell if it falls under the following condition:

$$ATFD > FEW \cap LAA < \frac{1}{3} \cap FDP \leq FEW$$

Where:

- ATFD: Access To Foreign Data metric; this metric measures whether the method is using a lot of attributes from other classes. If this is the case, then it is a sign that this method envies others.
- LAA: Locality of Attribute Accesses metric; the used attributes from other classes are far more than the used attributes from the method's own class. This is another sign that the method, or part of its functionality, is in the wrong place.
- FDP: Foreign Data Providers metric; the foreign attributes used by the method belong to very few other classes.
- FEW: denotes a "few" and was assigned the value 5.

Another approach to detecting bad smell is by performing a manual review of the code. Doing code reviews is considered the most reliable and accurate approach to detecting bad smells [1]. However, this comes at a price; it takes more time and it can't be repeated [9]. In addition to that, automatic detection of bad smells have been compared with manual approaches in many studies and have been found to be a suitable substitute [6, 9, 10]. Also, they can scale to large datasets much better than the manual approaches.

## 2.4 Maturity Models

Maturity models can be perceived as tools to evaluate the maturity of software process or products. As it is well-known, software products and processes cannot be perfect from day one. Maturity models can be used to measure the progress of an organization toward the perfect product or process. Models can be wide to cover more applications, like CMMI. However, such models need to be customized to the organization's needs to be useful [11].

Maturity models are designed as a set of progressive levels of effectiveness, where the organization is striving to move from its current level to the next level. The project or process is thought to be passing these maturity level as the product or process becomes more mature and as the organization becomes more capable. For an organization to apply a maturity model to its projects or processes, they need to start by assessing and determining their current level and figure out the capabilities they need to move to the next level.

We can think of maturity models as a way of prioritizing the capabilities needed to achieve excellence in a complex and long process. It acts as a guide for the organization to rank the capabilities to focus on what is most important right now. If you assessed your organization and concluded that it was on level 3 of some maturity model, then it makes sense that you focus on learning and applying the capabilities of level 4 instead of worrying about the capabilities of level 5 or any level beyond that.

Most of the new contributions in the field of maturity models are adaptations to existing well known models [12]. Most adaptations are using ISO/IEC 15504 or CMM/CMMI framework as a basis for their development. The adaptations are either: (1) an improvement of the original version of the maturity model this adaptation is based on, or (2) a custom-tailored version of the original

maturity model to be applied to specific types of domains and settings. Some of these domains that were the focus of many adaptations are: Security engineering service oriented domain, Testing/Assurance domain, eXtream Programming XP, Requirements, and more. In general, maturity models and frameworks are still advancing in terms of their coverage and the maturity of the model itself.

## 2.5 Prioritization Models

While analyzing, developing and testing software project, developers and project members usually face a situation where they need to pick an option from multiple possible options. These decisions varied from one case to another; stakeholders might need to select the next requirement or set of requirements to be implemented in the next development cycle, or it might be that developers need to select the most important test cases to be run.

If we took requirement prioritization as an example, we can assume that not all requirements are of the same importance to the customer. Hence, we need to prioritize the requirements based on many factors that are of interest to us; like customer satisfaction, time, and budget. Although this may look trivial if you have one client, consider a case where you need an input from different stakeholder who play different part in the project lifecycle and who may have different and sometimes contradicting priorities.

One simple approach to prioritizing requirements is the Priority Groups approach [13, 14]. In this technique, there are different priority groups that are different from each other based on their importance. There can be three groups named (low, medium, high), or it can be five groups named from 1 to 5, where 5 is of most importance. The next step is to assign requirement to groups based on their priorities.

In layman's terms, we consider prioritization models as processes that aim to rank different options systematically based on some criteria. It may take input from the relevant parties (customers, managers, developers) and along with the options to be ranked as input; it will give a weight to each option based on its importance according to the criteria.

# CHAPTER 3

# LITERATURE REVIEW

This chapter reviews the literature in two main topics: bad smell effect on software quality, and prioritization models. The first section discusses the literature that focused on the relationship between bad smells and software quality. The second section presents a survey of prioritization models in the field of software requirement, testing and maintenance.

## 3.1 Bad Smell Effect

One of the early work was done by Monden et al. [15] who studied the relationship between code clones and software reliability and maintainability. Using a legacy system, they found that small modules with clones are more reliable than modules without clones. However, when the clone codes get bigger, their modules became less reliable than non-clone modules. As for maintainability, they found that modules with clones are less maintainable than non-clone modules.

However, Kapser and Godfrey [16] identified several pattern of cloning and discussed their advantages and shortcomings. They also performed a case study which concluded that 71% of the clones are considered to have good impact on software maintainability.

Geiger et al. [17] studied the relationship between clones and change coupling; i.e. concurrent changes to files having the clones over time. They proposed a framework to extract clones, extract change coupling, and computing and visualizing the relation between clones and change coupling. They used the open-source Mozilla to validate their framework. Although the results show some

cases where a relation between clones and change coupling exists, there were no statistical correlation between the two.

Lozano et al. [18] developed a tool called CloneTracker, that tracks the changes (number of changes and density of changes) of methods when they contain clones and when they do not. What is special about this tool is that it detects clones at a finer level of granularity; the method level. The framework proposed at [17] focused on the file level. Doing a finer-grained analysis is beneficial in the case of clones, because most of them are introduced at the method level [19]. To test the tool, a case study was performed on the open-source system DnsJava was to either proving or disproving that clones are harmful. They concluded that methods change more often when they contain cloned code.

Saha et al. [20] studied how clone groups evolve with the evolution of the system. They extracted clone groups from 17 open-source systems ranging from Java (JUnit, JabRef, …) to the C family (Notepad++, 7-Zip, MAnt, ..). They found that most clones propagate through the different releases of the system without having any syntax changes. Also, they found that, on average, around 69% of clones reached the final release of the system. The authors hint that these clones may not have big effect on software maintenance since they do not require extra care. The results presented here expanded upon and agreed with a previous study conducted by Kim et al. [21].

Li and Shatnawi [22] studied the relationship between bad smells and class faults. They also associate bad smells with the severity of the faults. They studied six bad smells and used three version of the software Eclipse. They found that Large Class, Shotgun Surgery, and Large Method bad smells are significantly correlated with the severity levels of software faults, however, they

found that Data Class, Refused Bequest and Feature Envy bad smells are not significantly correlated.

Rahman et al. [23] conducted a study to validate whether clones result in more defects or not. They concluded that not only clones are not associated with defects, but that clone code is less defective than non-clone code.

D'Ambros et al. [24] studied the relationship between software defects and five bad smells. They found that even though some smells occur more than others; none of them can be more related to defects than others.

Marinescu and Marinescu [25] went further to answer weather classes that use classes with bad smells are more defect-prone than classes that do not use bad-smelled classes. They found that classes that use bad-smelled classes have more defect than other classes. They also found that the number of used bad smelled-classes has no effect on the defect-proneness of a class.

Khomh et al. [26] investigated the impact of bad smells on change-proneness. With 9 releases of Azureus and 13 releases of Eclipse, they investigated the relationship between code smells and change-proneness and found that the possibility of change is higher if the class contains bad smells. They also empirically validated that a class with a higher number of bad smells is more likely to be a change-prone class. They also showed that some bad smells are more related to change-proneness than others.

Olbrich et al. [27] studied the effect of God class and Brain class with respect to three measures: change frequency, change size, and defect. They investigated both smells in three open-source systems. Their experiment concluded that God and Brain classes are more prone to change and defects. These findings were in agreement with a previous study by the authors [28]. However,

when the classes were normalized with respect to size, the results were reversed. God and Brain classes were less likely to change and have defect. This suggests that the existence of these classes may have a positive impact on the quality of the system as a whole.

Deligiannis et al. [29] performed an experiment to study the effect of God Class smell on the understandability and maintainability of the system. A group of 20 subjects were asked to perform a maintenance task and answer questions regarding two versions of the system. Both versions of the design capture the same functionalities. The first version, as shown in Figure 2, is the design compliant version of the system. The second version, as shown in Figure 3, is a modified design non-compliant version of the system where a god class was introduced. They found that design compliant systems are easier to maintain than design non-compliant systems.

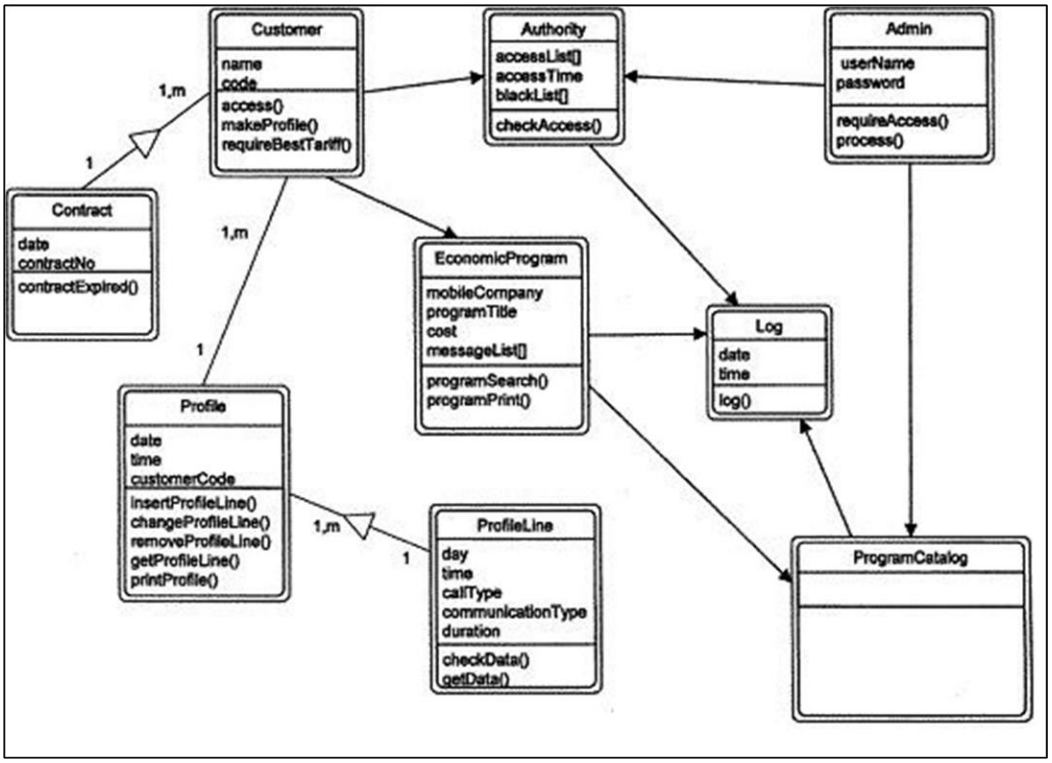



**Figure 2: A design compliant version of the system under study in [29]**

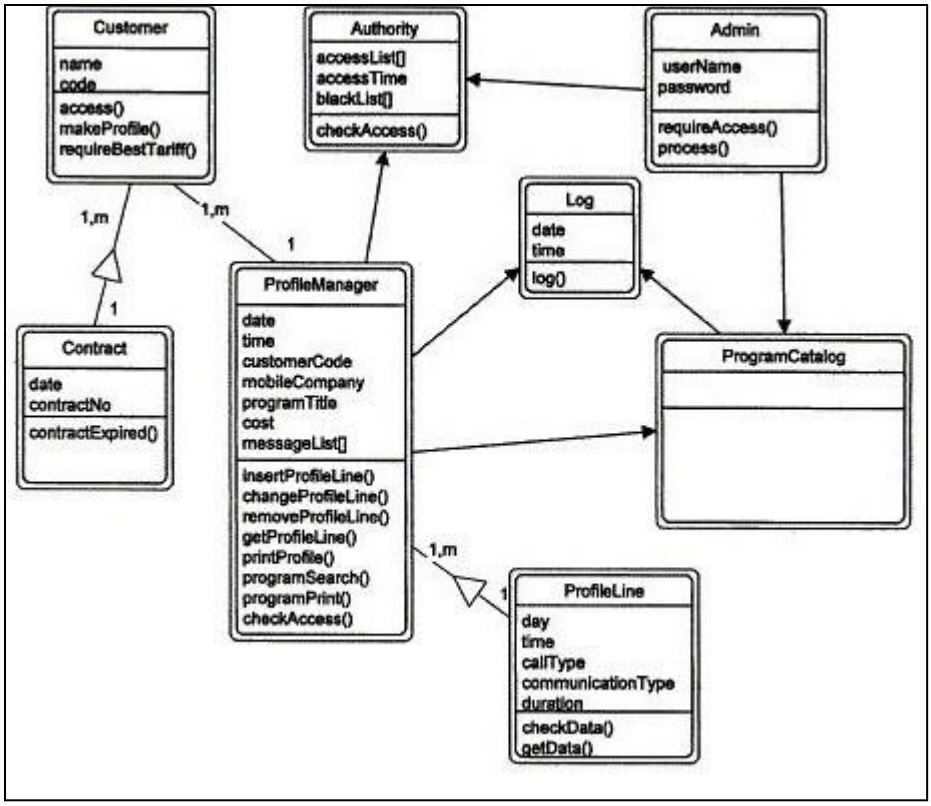



**Figure 3: A modified version of the system design where a God Class (ProfileManager) was introduced [29]**

Yamashita and Moonen [30] performed an experiment where subjects recorded problems surfacing during the maintenance of system files and then tried to see if code smells detected in these classes could predict the recorded problems. They found that code smells are partial indicator of problems during maintenance. Yamashita and Counsell [31] expanded the previous work and tried to see if code smells could be considered as maintainability indicator at the system-level.

Table 3 summarizes the literature on code smell effect on different aspect of quality attributes. It also compares our research to the literature. The following criteria were used: bad smells considered, quality attributes investigated, data (or subjects) used in the experiment, the type of the experiment (empirical, case study), the level of the discussed bad smells (code or design), and the threshold used to determine the existence of the bad smell.

**Table 3: A summary of the literature compared to this work.**

| | Bad Smells | Quality Attributes | Data/Subjects | Experiment Type | Level | Threshold |
|---|---|---|---|---|---|---|
| Monden, et al. (2002) [15] | Clones | Reliability Maintainability | Legacy System | Empirical | code | 30 same-line |
| Kapser and Godfrey (2008) [16] | Clones | Maintainability | Apache httpd Gnumeric | Case study | Code | - |
| Geiger, et al. (2006) [17] | Clones | Change coupling | Mozilla | Case study | File | 30 tokens using CCFinder [32] |
| Lozano, et al. (2007) [18] | Clones | Number of changes Density of changes | DnsJava | Case study | Method | Using CCFinder [32] |
| Saha, et al. (2010) [20] | Clones | - | 17 open-source systems | Empirical | Code | Using CCFinderX [33] |
| Li and Shatnawi (2007) [22] | Data Class God Class God Method Refused Request | Error probability Error severity | Eclipse | Empirical | Code | See [34] |
| Rahman, et al. (2012) [23] | Clones | Defect-proneness | Apache httpd Nautilus Evolution Gimp | Empirical | Code | 50 same-lines using DECKARD tool |
| D'Ambros, et al. (2010) [24] | Brain Method Feature Envy Intensive Coupling Dispersed Coupling Shotgun Surgery | Software defects | 6 java-based systems | Empirical | Code | See [35] |
| Marinescu and Marinescu (2011) [25] | Data Class God class Brain class Feature Envy | Software defects | Eclipse | Empirical | Code | See [36] |
| Khomh, et al. (2009) [26] | 29 code smells | Change-proneness | Azureus, Eclipse | Empirical | Code | See [1] |

[1] http://www.ptidej.net/downloads/replications/wcre09a/

| | | | | | | |
|---|---|---|---|---|---|---|
| Olbrich, et al. (2010) [27] | God class<br>Brain class | Change frequency<br>Change size<br>Weighted defects | Lucene<br>Xerces<br>Log4j | Empirical | Code | See [36] |
| Deligiannis et al. (2004) [29] | God Class | Maintainability | 20 undergrad students | Controlled Experiment | Design | - |
| Yamashita and Moonen (2013) [30] | 12 Code smells | Maintainability | 6 professional developers | Case study | Code | - |
| Yamashita and Counsell (2013) [31] | 12 Code smells | Maintainability | 6 professional developers | Case study | Code | - |
| Our Work | 5 code smells | Maintainability | Apache Ant, ArgoUML, and log4j | Experiment | Code | See [37] |

## 3.2 Prioritization & Maturity Models

Iqbal et al. [38] proposed a prioritization model of requirements for market driven software products. In market driven products, managers face huge challenge to select the appropriate requirements from the enormous lists that comes from users, focus groups, interviews, and competitions product's analysis. The objective of this model is to provide software managers with a way to select the most important and relevant requirements. The proposed model, called Market Driven Requirement Prioritization Model (MDROM), is a four-step model and uses the Analytical Hierarchical Process (AHP) [39]. The first step is to *analyze every requirement by software analysts* and assign it one of 31 pre-defined categories called bins. Some of these bins are: Accessibility, Availability, Usability, and Security. Second step is to process each bin's requirement in AHP engine [39]. This step involves assigning weights to each requirement according to its importance in relation to other requirements. Let's assume that we have 'n'

requirements, $R_1$, $R_2$, …, $R_n$. We put them inside nxn matrix. For each pair $R_{ij}$, we assign a weight representing how much importance $R_i$ is in relation to $R_j$. For example, $R_{11}$ will have a weight of 1, obviously. If we assign $R_{23}$ a weight of 2, this means that $R_2$ is twice as important as $R_3$. At the end, the matrix will be normalized by column and by row to get the final comparison matrix. Step three is where consist checks are performed. The last step is where we prioritize the bins using the previous two steps

Perini et al. [40] adopted machine learning techniques to come up with a requirements prioritization model. The proposed model, called Case-Based Ranking (CBRank), uses an ordering framework introduced in [41] and [42]. What is unique in this proposed method is that it mixes stakeholders opinion in requirements ordering with ordering measured by machine learning. The process is in an *iterative* three steps process: (1) Pair Sampling: a set of sampled requirements pairs, whose orders are not known yet, is selected from the Requirements pool. (2) Priority Elicitation: the decision makers order the pairs. (3) Priority Learning: is where the learning algorithm generates an Approximation Rank of Requirements based on the previous iterations. This iteration stops when the result of the last step is considered accurate or time to elicit and rank more pairs runs out. When it stops, the output of the final step of the final iteration (the Approximation Rank) is the Final Approximation Rank, which will be used to prioritize requirements. This method reduces the time needed for a human to input his preferences. It also gives a more accurate estimate of the final ranking. It also allows its user to decide when to stop the iteration by assessing the tradeoff between the elicitation efforts needed and the accuracy improvement.

Saleh et al. [43] proposed a three-domain prioritization framework for preventive maintenance of medical equipment. The first domain is where you identify customer needs (Voice of customers)

and technical characteristics (Voice of Engineers). they adopted a simplified version of house of quality (HOQ) matrix [44]. The relationships between Customer and technical requirements are indicated by a 1-9 score. The next domain is where you identify the critical criteria from the previous domain and divide them into three categories: risk-based, mission-based, and maintenance-based. Then, using HOQ matrix, and with the authors experience as an input, a new matrix is calculated for the critical criteria. The final domain, the concept domain, is a priority score equation: the sum of the critical criteria multiplied by the criteria score. Using this equation, one can calculate the priority score of a maintenance task by knowing which critical criteria are related to it and what is the assigned scores for such criteria.

Kavitha et al. [45] proposed a Test Case Prioritization (TCP) model *based on software Requirements specification*. The proposed model starts with prioritizing requirements using three scores: (1) Customer Priority (CP): a 1-10 value assigned by the customer and measures the importance of the requirement in his opinion. (2) Implementation Complexity (IC): A 1-10 value assigned by the developer indicating the perceived complexity of implementing the requirement. (3) Requirement changes (RC): the number of times a requirement has been changed, normalized to a scale of 10. The weight of a requirement is the average weight of the three scores above. Next step is to map test cases to requirements. Then, each test case is assigned a weight, which is the fraction of requirement weight of the test case map divided by the total requirements weight. The order of test case running is decided based on the test case weight. The proposed model has proven to be more effective in fault detection in comparison to randomized test execution.

Alves et al. [46] also proposed a test case prioritization model. Their model is, however, based on refactoring activities performed on the system under test (SUT). The goal is to avoid the need to re-run the whole regression suite after refactoring. Their approach is as follows: First, two version

of the SUT, before and after refactoring, is needed. Second, types and locations of refactoring edits are identified using a support tool. Then, code elements (methods) affected by the refactoring edits are identified. In addition, a call graph for each test case is generated (to know which affected methods are covered by a given test case). Then, a test selection is performed where every test case whose call graph includes any of the affected methods is selected. If prioritization is not needed, the output of this step (the selection set) could be used to test the system after refactoring. To prioritize the test cases, we assign an impact value to each test case. The value represents the number of affected method in the test case's call graph. The higher the number of methods in the call, the higher the importance of the test case is. Test cases that are not part of the selection set are assigned an impact value of 0. The model proved to be more effective in producing more stable systems than traditional prioritization models.

Correia and Visser [47] proposed a certification method for the quality of software products. They used a layered approach using the ISO-9126 standard and focused on maintainability. The layered model was used to measure and rate the software quality. The model uses metrics to understand the system properties, which then are mapped and related to the sub-characteristics as shown in Figure 5, and then the main characteristic; i.e. maintainability.

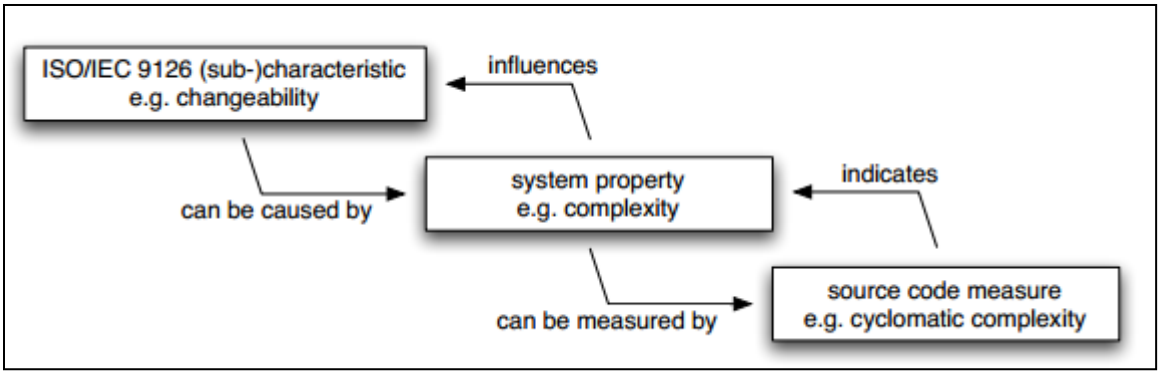



**Figure 4: The relation between source code metrics and system sub-characteristics of maintainability, as established by the SIG model**

For example, to calculate the Maintainability value, you find the sub-characteristic values and average them. For each sub-characteristic value, you have properties' values that can averaged too. Properties can be measured by simple metrics.

properties

| ISO 9126 maintainability | duplication | test quality | unit complexity | unit size | volume |
|---|---|---|---|---|---|
| analysability | × | | | × | × |
| changeability | × | | × | | |
| stability | | × | | | |
| testability | | × | × | × | × |

**Figure 5: The relation between sub-characteristics and system properties**

Then, the authors proposed a three steps appraisal method to certify software. Every software could be certified at one of five possible quality levels, based on its score.

Baggen et al. [48] described a software certification process developed by the Software Improvement Group (SIG) and is focused on maintainability. They used ISO-9126 standard to define maintainability. SIG selected six source code properties as key indication of quality and they can be calculated at the unit level:

- **Volume:**

  The larger a system, the more effort it takes to maintain.

- **Redundancy:**

  Duplicated code has to be maintained in all places where it occurs;

- **Unit size:**

  Units as the lowest-level piece of functionality should be kept small to be focused and easier to understand;

- **Complexity:**

  Simple systems are easier to comprehend and test than complex ones;

- **Unit interface size:**

  Units with many parameters can be a symptom of bad encapsulation;

- **Coupling:**

  Tightly coupled components are more resistant to change.

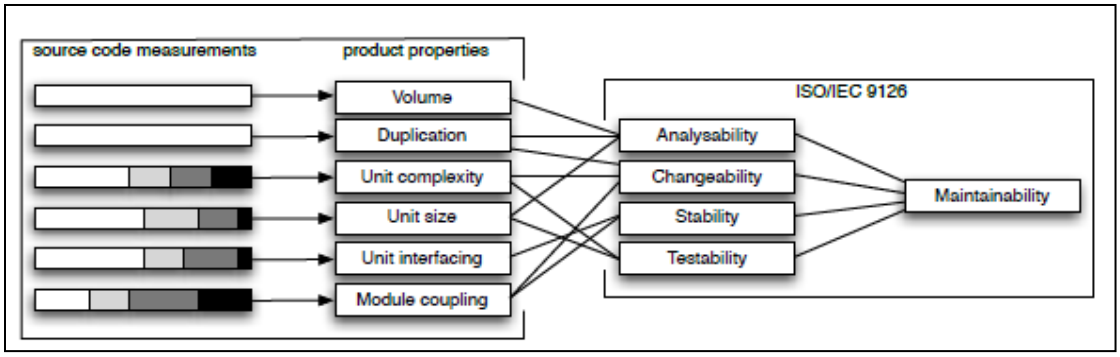



**Figure 6: The relation between source code metrics and sub-characteristics of maintainability**

# CHAPTER 4

# RESEARCH METHODOLOGY

In order to address the research questions proposed earlier, we propose the use of maturity models to gauge the effect of removing bad smells from the system. The key steps in our research methodology are outlined in Figure 7 and the main components will be discussed in the following sections.

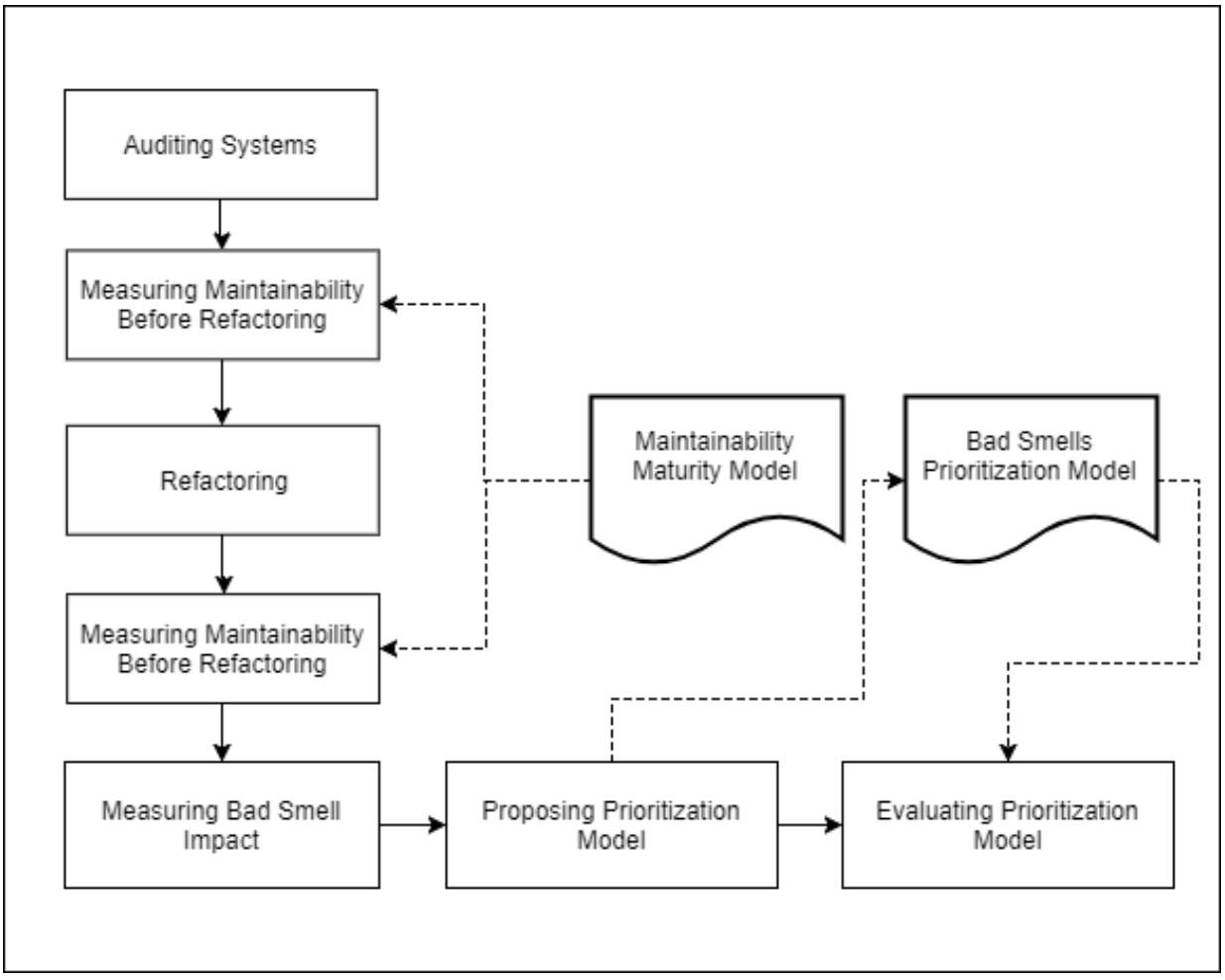



**Figure 7: A representation of the proposed research methodology**

## 4.1 Auditing Systems

In this step, the systems under study will be subject to a Bad smell audit to identify the bad smells in the system. Performing a manual audit might be feasible depending on many factors like: size and complexity of the system under audit, time available to perform the audit, familiarity with the concept of bad smell and refactoring, the number and types of bad smells that are being investigated, and the expertise of the auditor.

However, this step is usually done with the aid of an automated tool. Automated tool will help in detecting possible bad smells in the system. This is faster and easier to perform, especially when you have large set of classes and you are not familiar with them.

The down side of this approach is the accuracy of the detection technique. Not every bad smell possibility reported by the tool can be refactored. Sometimes the tool reported a false positive. Sometimes the bad smell cannot be removed because the benefit of removing it will have a negative impact on the system quality. For example, let's say you have a class that has some heavy GUI operations that uses many public fields. Those public fields can be encapsulated. Hiding those fields behind an accessor might impact the performance and speed of the GUI operations

## 4.2 Measuring Maintainability

To measure the Maintainability of the system, of the system, we propose a Maintainability Model that is described in the next section. The model will use a metric-based approach to evaluate the maintainability of the system.

Of course, to measure the impact of removing bad smell from the system on the system maintainability, you need to have a point of reference. Which means that you need to analyze the

system before and after refactoring bad smells. We used an automated tool to calculate the relevant metrics to our research.

The process of measuring maintainability before refactoring is done once for each system. However, the process of evaluating maintainability after refactoring is done once for each pair of bad smell and system, as explained in the previous section. This will help us measure the impact of a certain bad smell without having to worry that the refactoring of other bad smells might affect its results. Each dataset that was the results of the refactoring process will be compared to the original dataset of the system; i.e. the maintainability of the system before refactoring.

## 4.3 Maturity Model for Maintainability

We will use a modified version of the quality model proposed by Baggen et al. [48]. This model will be used in the measuring software maintainability before refactoring and after refactoring, where we re-evaluate the system using the maturity model after applying the refactoring. One of the main motives to choose this maturity model is that it has been updated to confirm to the newer ISO 25010 definition of Maintainability. ISO 25010 defines Maintainability as the: "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [49]. Maintainability is divided into five sub-characteristics which are defined as follows [49]:

- **Analyzability**: The degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

- **Modifiability**: The degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality

- **Testability**: The degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

- **Modularity**: The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- **Reusability**: The degree to which an asset can be used in more than one system, or in building other assets

Since our focus is in the class level, we modified the model to reflect that. The main components of the model are described in Figure 8. The relationship between Maintainability's sub-characteristics and the system properties used in this research is described in Table 4. These relationships are based on the work done in [48, 50]. In the context of this research, the relevant system properties are:

- Volume: The overall size of the class source code. As the size of the class increases, its analyzability will be affected making it harder to be understood. Also, large classes are harder to test than others and require more effort in terms of writing test cases.

- Duplication: The degree of duplication in the class .Duplication is defined as the occurrence of identical fragments of source code in more than one place in the product . Having duplicate in the class will affect its analyzability effort. Also, the more duplications

in the system the harder it is to modify these duplicated fragments since they are scattered and repeated across the system. On the other hand, if those duplicated fragments were extracted in one place, the modifiability will be much easier.

- Unit Complexity: The degree of complexity in the class's unit; i.e. methods or functions. Having a complex fragment of code will increase the effort needed to modify it. Also, the effort and time to test it will increase.

- Unit size: The size of the classes units; i.e. methods or functions. Having a reasonable unit size will improve the analyzability of those unit. It also, will ease the process of reusing those units.

- Unit interfacing: The size of the interfaces of the units of the class. Having large interfaces of your methods will hinder the ability of reusing those methods on other parts of the system. The reason is that a unit with a high number of parameters requires you to have more knowledge about the unit's parameters and the type of values expected to reuse it.

- Coupling: The coupling between classes in terms of the number of incoming dependencies. Coupling does affect the modifiability because a change to class that is used by other classes will require a change to the other classes. Coupling affect modularity also for the same reason.

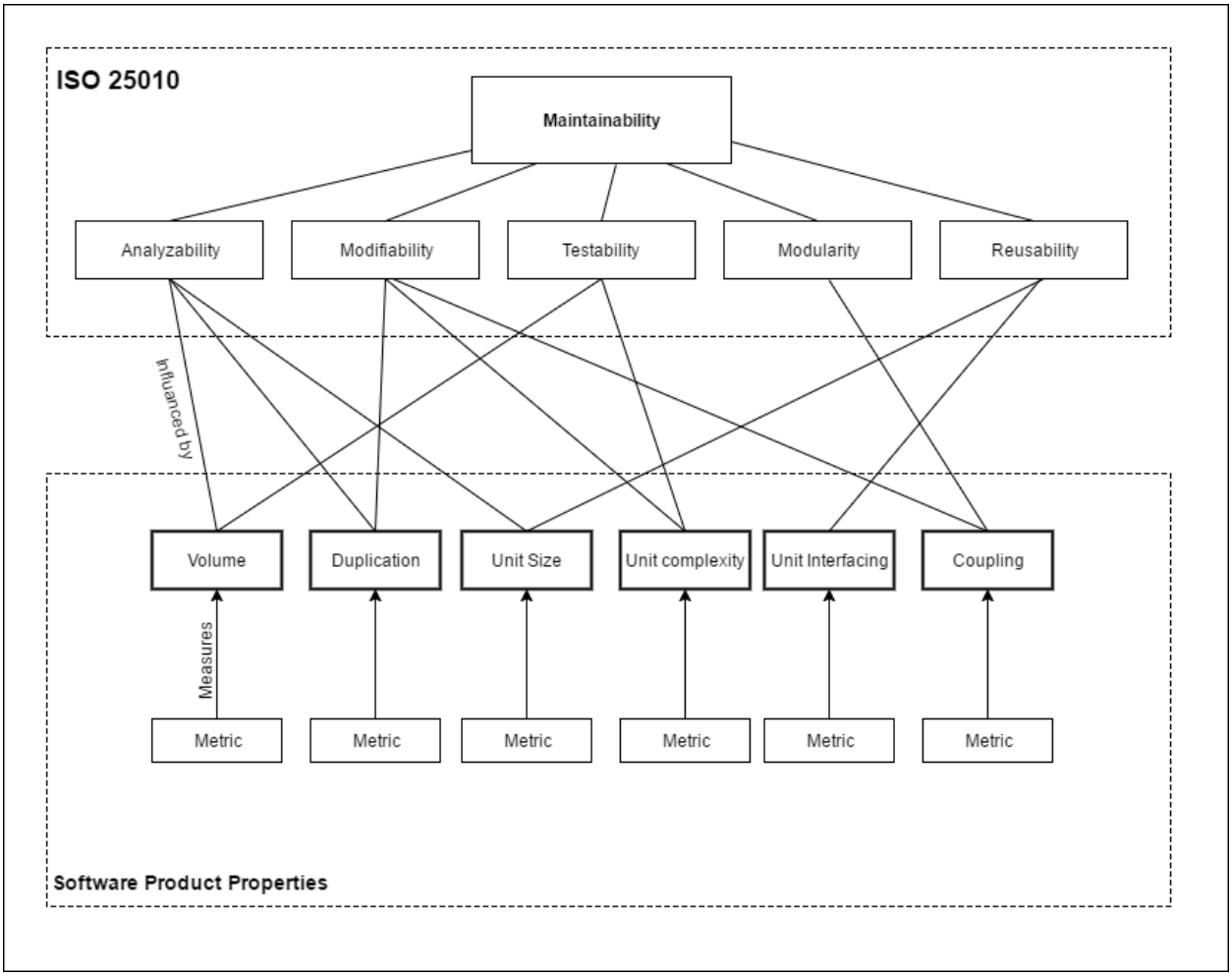



**Figure 8: The proposed Maintainability Model**

As described in Figure 8, each software product property is measured by a software metric. The value of this metric will indicate the state of that product property in the software. For example, we can use Lines of Code metric to estimate the value of the Volume property in the software.

Our next step is to calculate the values for every sub-characteristic of Maintainability. We can do that by aggregating the values of the properties that influence each sub-characteristic. For example, to calculate the value of the Analyzability sub-characteristic, we can aggregate the values of the Volume, Duplication, and Unit Size properties.

Finally, we need to quantify the Maintainability of the software by assigning a value to it. In our model, we will use the five sub-characteristics of Maintainability to estimate its value. By aggregating the values for Analyzability, Modifiability, Testability, Modularity and Reusability we can have an estimate of the Maintainability. This estimate will be used by the next steps in our research to quantify the impact of bad smells on maintainability, and to evaluate the prioritization model that will be proposed later.

**Table 4: Relationship between ISO 25010 Maintainability's sub-characteristics and System properties.**

| | Volume | Duplication | Unit size | Unit Complexity | Unit Interfacing | Coupling |
|---|---|---|---|---|---|---|
| **Analyzability** | X | X | X | | | |
| **Modifiability** | | X | | X | | X |
| **Testability** | X | | | X | | |
| **Modularity** | | | | | | X |
| **Reusability** | | | X | | X | |

## 4.4 Refactoring

This step involves analyzing and refactoring the bad smell reported for each system audit in Initial Audit stage. First, the reported bad smell possibilities are analyzed to see if it makes sense to refactor them. As described earlier, not all reported bad smell possibilities are good to refactor. You need to analyze each instance and see if it is worth refactoring, and if the bad smell was introduced on purpose to achieve a certain improvement in quality.

Second, for each bad smell type and each system, a refactoring process is carried out by applying any of the possible refactoring methods to remedy the bad smell. The result code of this process is saved to be analyzed later. This step will repeat for each system and each bad smell type detected

in the system. For example, let's say you have System A with Bad Smells BS X and BS Y and you have System B with BS X. Then, you will do three refactoring processes: refactor BS X in System A, BS X in System B, and BS Y in System A. Which means you will have three different sets of source code that will be analyzed later.

To insure the correctness of the refactoring process, we make sure that there are no errors raised by the IDE after doing the refactoring. As mentioned earlier, refactoring is altering the internal structure of the code without affecting its external behavior. However, we couldn't compile and run each project under study because of many factors, like its dependency on other libraries and projects, and time constraints.

## 4.5 Measuring Bad Smells Impact

After we evaluate the systems before and after the refactoring process using the maturity model, we can now study the impact of bad smells on maintainability. This is mainly what we will discuss in the next chapter, where we formalize our experiment and present its results. In this step, each bad smell is discussed separately, where the change in the maintainability values for the classes that were affected by the refactoring of said bad smell will be presented and discussed.

The goal in this step is to observe a common change trend among different classes of different systems after refactoring for the same bad smell, and to explain any contradictions if present. So, if refactoring for a bad smell resulted in a positive change in maintainability, we can conclude that refactoring this smell will improve the maintainability. However, if classes reported contradicting data, i.e. some improve and some do not, further analysis will be done to explain the difference.

## 4.6 Proposing and Evaluating the Prioritization Model

This step involves proposing a prioritization model, and evaluating the model using the data from the previous step. The Bad Smells Prioritization Model (which will be presented in detail in Chapter 6) aims at prioritizing bad smells based on their impact on maintainability. There are different steps involved from calculating impact to scaling the impact and applying a pair-wise comparison. The final result of this step is an ordered list of bad smells based on the bad smells' impact on the maintainability of the software.

# CHAPTER 5

# EMPIRICAL EVALUATION

In the following sections, the design of the experiment is discussed in detail. The Context section is provided, and the Hypothesis are formulated.

The goal of this experiment is to observe and validate the impact of refactoring bad smells on maintainability. In the results section, each bad smell will be discussed separately. For each bad smell, the change of classes' maintainability is reported and discussed. Finally, the results of this experiment will be used in the next chapter to evaluate the proposed prioritization model.

## 5.1 Context Selection

The context of an experiment is a great way to help researchers streamline the process of replicating the experiment. It saves the time and effort required to setup the experiment and run it. The context of this experiment is three open source Java systems. Hence, the experiment is specific which may hinder the ability to generalize the experiment's results. Furthermore, the experiment investigates a subset of bad smells: Feature Envy, Shotgun Surgery, Large Method, Large Class, and Data Class. Further discussion of the threats to validity will be presented in later sections.

## 5.2 Hypothesis Formulation

Our goal is to investigate the effect of bad smells in the maintainability of classes. For each Bad Smell under study, we will analyze the classes before and after the removal of that bad smell; i.e

refactoring. We will use metrics to predict the maintainability change in the class after refactoring. To focus our study, we will set up the following hypothesis:

- $H1_0$: The maintainability of a class is not affected by refactoring Feature Envy. We can define the alternative Hypothesis $H1_a$ as: The maintainability of a class is affected by refactoring Feature Envy

- $H2_0$: The maintainability of a class is not affected by refactoring Shotgun Surgery. We can define the alternative Hypothesis $H2_a$ as: The maintainability of a class is affected by refactoring Shotgun Surgery.

- $H3_0$: The maintainability of a class is not affected by refactoring Large Method. On the other hand, the alternative Hypothesis $H3_a$ is defined as: The maintainability of a class is affected by refactoring Large Method.

- $H4_0$: The maintainability of a class is not affected by refactoring Large Class. The alternative Hypothesis $H4_a$ can be defined as: The maintainability of a class is affected by refactoring Large Class.

- $H5_0$: The maintainability of a class is not affected by refactoring Data Class. The alternative Hypothesis $H5_a$ is defined as follows: The maintainability of a class is affected by refactoring Data Class.

## 5.3 Variable Selection

As defined in the Maturity Model used in this research, the dependent variable is the change in Maintainability. This value will be calculated based on the proposed model in the previous chapter.

A positive value means an improvement in the class maintainability. A negative value indicates a decrease in the class maintainability. The independent variables are the following selected metrics:

- **Logical Line of Code (LLOC):**

  The number of code lines of the class, including the lines of its local methods. Empty and comment lines are not counted.

- **Clone Coverage (CC):**

  The ratio of code covered by code duplications in the class to the size of the class.

- **Sum of methods' McCabe Cyclomatic Complexity (Sum of McCabeCC):**

  The sum of the number of independent control flow paths in a class's methods.

- **Sum of methods' LLOC (Sum of LLOC):**

  The sum of code lines of the class's methods. Empty and comment lines are not counted.

- **Sum of number of methods' parameters (Sum of NUMPR):**

  The sum of number of the parameters of the methods in a class.

- **Number of Incoming Invocations (NII):**

  The number of other methods and attribute initializations which directly call the local methods of the class.

Table 5 maps each one of the above metrics to a software property in the model.

**Table 5: The mapping between software product properties and software metrics used in the experiment.**

| Software properties | Software measurement used |
|---|---|
| Volume | Logical Line of Code (LLOC) |
| Duplication | Clone Coverage (CC) |
| Unit Size | Sum of methods' LLOC (Sum of LLOC): |
| Unit Complexity | Sum of methods' McCabe Cyclomatic Complexity (Sum of McCabeCC) |
| Unit Interfacing | Sum of number of methods' parameters (Sum of NUMPR) |
| Coupling | Number of Incoming Invocations (NII) |

## 5.4 Research Data

We use five open source systems for this research, Apache Ant, ArgoUML, Log4j, JHotDraw, and Xerces-J. Some Descriptive data about the system are presented in Table 6.

**Table 6: Systems used in the research.**

| System | Size LOC | # of classes | # of methods |
|---|---|---|---|
| Ant | 225690 | 1204 | 10804 |
| ArgoUML | 124922 | 808 | 6261 |
| Log4j | 54671 | 361 | 2523 |
| JHotDraw | 46888 | 416 | 3389 |
| Xerces-j | 205878 | 728 | 8452 |

Apache Ant [51] is a command-line tool that asses in the build of applications. Ant provides built-in tasks that will help in the process of assembling, running, and compiling application. Although it is primarily used for Java application, Ant cannot be used in other programming languages like C++.

ArgoUML [52] is java-based open-source UML modeling software. It has two main features: (a) the ability to create and build UML diagrams, and (b) it can reverse engineer an existing Java code to generate UML diagrams based on it.

Log4j [53] is a logging library for Java applications. It is maintained by the Apache Software Foundation [2] and is one of the popular logging libraries for Java.

JHotDraw [54] is an open source GUI framework for Java language. Although it was initially developed for exercise, it became a well-established and quite powerful in the world of structured drawing editors.

Xerces [55] is a set of libraries specialized in validating, parsing and manipulating XML files by utilizing a very easy to use framework. The framework has implementations in C++, Java, and Perl. The Java implementation is used in this experiment.

## 5.5 Selected Bad Smells

The following five bad smells are used in this research. They represent different groups of bad smells as presented in [5]. Namely, the selected Bad Smells are from five of the seven categories: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, and Couplers. The five bad smells are:

- **Feature Envy:**

  This bad smell represents methods that are using data from other classes more than they are using their own class's data [1].

[2] https://www.apache.org/

- **Shotgun Surgery:**

  This bad smell tells you that a change in a class may lead to a lot of changes in other classes due to the fact of strong coupling [1].

- **Large Method:**

  This bad smell is a sign that a method is doing most of the heavy lifting in the class [1]. It may be the case that this method started as a small one, and then many responsibilities were assigned to it until it reaches a point where it is difficult to maintain.

- **Large Class:**

  This bad smell is a sign that a class is playing a centralized role in the system with little delegation. These types of classes tend to do most of the work and use data from other classes [1]. Such Large Classes are harder to understand and maintain.

- **Data Class:**

  A Data Class is a class that mainly holds data with no to little functionality [1]. This may be a sign that the functionalities performed on the class's data are distributed among other classes and it may be best to move these functionality to the class.

## 5.6 Data Analysis

To compare the effect of bad smell on class maintainability, we need two sets of metric data: data before refactoring the code, and data after refactoring. Firstly, we collected metric data for each

system under study using SourceMeter tool [3]. SourceMeter is a command-line static analysis tool. It supports Java, Python, C, C++, and PRG. The tool analyzes the software in different level of abstractions. The tool generates relevant metrics for packages, files, classes, methods, attributes and more. The tool supports a long list of software metrics in addition to providing an easy integration with other popular tools like PMD and FindBugs.

As presented earlier, our maturity model requires five software metrics for each class. We extracted the required class metrics, and aggregated the required method metrics because our analysis is for the class-level.

Then, we analyzed the systems for bad smells using Borland Together [37]. Together is a modeling tool to understand and develop the software architecture. One of the main feature in this tool is the ability to perform an audit on the software to discover problems and style and rule violations. It also can detect a list of code bad smells and report the result in an easy to understand way. It comes as a plugin to the infamous Eclipse IDE.

The tool produced a list of *possible* bad smells' instances in each system. Then we analyzed each reported bad smell to see if it is in fact a bad smell or a false positive, and to see if refactoring the bad smell will make sense or not. Then, for each pair of system and bad smell, we did the refactoring of existing bad smells in the system. After refactoring a system for a bad smell, we collected the five software metrics for the refactored classes again.

These five metrics, as discussed, will be used to gauge the maintainability of the class, i.e. use the maturity model to give each class a maintainability value. Because we have two sets of values for each class (before and after refactoring for a bad smell) the difference will be an indicator of the

[3] https://www.sourcemeter.com/resources/

impact of said bad smell on the maintainability of the class. This difference, or what we call a maintainability change, will be observed and discussed for each bad smell in the following sections.

## 5.7 Research Results

In the following sections, we will discuss the results observed for each bad smell under study. For each bad smell, we will present the change in maintainability for each class and comment on the results. The change in maintainability was calculated by subtracting the maintainability values of the class after refactoring from the value before refactoring. The maintainability value of the class was calculated by averaging the values for each of its sub-characteristics. More details are presented in Section 04.3.

### 5.7.1 Feature Envy

Table 7 presents the number of bad smells instances detected and reported by the tool in each system, and the number of instances that were verified to be bad smells and, in turn, refactored. Table 8 list the classes affected by the refactoring of Feature Envy bad smell and their maintainability value change.

**Table 7: Summary for Feature Envy Audit.**

| System | Detected Bad Smell Possibilities | Refactored Bad smells instances |
|---|---|---|
| Ant | 1 | 1 |
| ArgoUML | 4 | 3 |
| Log4j | 1 | 1 |

| | | |
|---|---|---|
| JHotDraw | 5 | 2 |
| Xerces-j | 1 | 1 |
| **Total** | **12** | **8** |

**Table 8: Comparison of the maintainability of affected classes before and after Feature Envy**

| System | Class | Maintainability | | |
|---|---|---|---|---|
| | | Before | After | Change |
| ant | DOMUtil | 39.333333 | 40.700000 | -1.366667 |
| argo | ActionAddNote | 24.600000 | 26.700000 | -2.100000 |
| argo | PropPanel | 187.600000 | 187.166667 | 0.433333 |
| log4j | LBELEventEvaluator | 18.166667 | 21.566667 | -3.400000 |
| log4j | Node* | 23.066667 | 23.600000 | -0.533333 |
| jhotdraw | TextTool | 33.79767 | 35.16364 | -1.365970 |
| jhotdraw | AbstractTool | 104.0909 | 100.9333 | 3.157600 |
| xerces | CMBuilder | 62.93333 | 58.66667 | 4.266660 |
| xerces | XSParticleDecl | 51.03333 | 55.56667 | -4.533340 |

**Observations:** eight instances of feature envy bad smell were refactored in the five systems. We found that the removal of Feature Envy code bad smell has a negative impact on the maintainability of the classes as clearly shown in Table 8. This also includes the one class indirectly affected by the refactoring process Node, which is denoted by a star (*) in the table.

However, for three cases, the maintainability increased. After investigation, we found that the refactoring method we used in classes PropPanel and AbstractTool, Extract Method, extracted a

duplicate code, which explains why the classes's maintainability increased. Moreover, we applied a different refactoring to classes CMBuilder and XSParticleDecl which was Move Method refactoring. We found that the maintainability increases in the class that loses a method (i.e. CMBuilder class) and decreases in the class that receives it (i.e. XSParticleDecl class). Based on that, we can reject the null hypothesis $H1_0$ and accept the alternative hypothesis $H1_a$.

### 5.7.2 Shotgun Surgery

Table 9 presents the number of Shotgun Surgery smells instances detected and reported by the tool in each system, and the number of instances that were verified to be bad smells and, in turn, refactored. Also, Table 10 list the classes affected by the refactoring of Shotgun Surgery bad smell and their maintainability value change.

**Table 9: Summary for Shotgun Surgery Audit.**

| **System** | **Detected Bad Smell Possibilities** | **Refactored Bad smells instances** |
|---|---|---|
| Ant | 8 | 1 |
| ArgoUML | 8 | 3 |
| Log4j | 2 | 1 |
| JHotDraw | 2 | 0 |
| Xerces-j | 3 | 0 |
| **Total** | **23** | **5** |

**Table 10: Comparison of the maintainability of affected classes before and after Shotgun Surgery.**

| System | Class | Maintainability | | |
|---|---|---|---|---|
| | | Before | After | Change |
| ant | JUnitTest | 102.96667 | 60.80000 | 42.16667 |
| argouml | ArgoEvent | 8.80000 | 10.80000 | -2.00000 |

| | | | | |
|---|---|---|---|---|
| argouml | Critic | 141.26667 | 139.43333 | 1.83333 |
| argouml | KnowledgeTypeNode | 10.53333 | 12.36667 | -1.83333 |
| argouml | ActionActivityDiagram* | 13.66944 | 13.67018 | -0.00073 |
| argouml | ActionAddAttribute* | 6.51044 | 6.50905 | 0.00139 |
| argouml | ActionAddOperation* | 6.51044 | 6.50905 | 0.00139 |
| argouml | ActionDeploymentDiagram* | 6.38222 | 6.38182 | 0.00040 |
| argouml | ActionSequenceDiagram* | 6.38222 | 6.38182 | 0.00040 |
| argouml | ActionStateDiagram* | 13.34200 | 13.34286 | -0.00086 |
| argouml | ActionUseCaseDiagram* | 6.38222 | 6.38182 | 0.00040 |
| argouml | UMLAction | 39.03333 | 49.16667 | -10.13333 |
| log4j | Util | 24.26667 | 25.26667 | -1.00000 |
| log4j | ConnectionSource | 3.333333333 | 2.333333333 | 1.00000 |

**Observations:** five instances of shotgun surgery bad smell were refactored in the investigated systems. We used the refactoring technique Move Field. However, we noticed a clear difference in class maintainability depending whether we move the fields to the class or we move them from the class. We noticed that the maintainability is improved when we move field from the class. When we move field to the class, however, the maintainability decreases. This suggests that the impact of refactoring bad smells is defendant on the type of refactoring method used.

For the classed indirectly affected by the changes, which are denoted by a star in Table 10, They all had slight improvement in maintainability, except for two classes: ActionActivityDiagram and ActionStateDiagram. After further investigation, we found that all the classes share a common parent class, UMLAction, and their implementation is trivial except for the two classes mentioned above. We can conclude that the difference in maintainability is attributed to the difference in the

classes volume and not the refactoring process. Based on the presented observations, the null hypothesis $H2_0$ is rejected and the alternative hypothesis $H2_a$ is accepted.

### 5.7.3 Large Method

Table 11 presents the number of Large Method smells instances detected and reported by the tool in each system, and the number of instances that were verified to be bad smells and, in turn, refactored. Also, Table 12 list the classes affected by the refactoring of the bad smell and their maintainability value change.

**Table 11: Summary for Large Method Audit**

| **System** | **Detected Bad Smell Possibilities** | **Refactored Bad smells instances** |
|---|---|---|
| Ant | 20 | 18 |
| ArgoUML | 2 | 2 |
| Log4j | 5 | 5 |
| JHotDraw | 0 | 0 |
| Xerces-j | 11 | 4 |
| **Total** | **38** | **29** |

**Table 12: Comparison of the maintainability of affected classes before and after Large Method.**

| System | Class | Maintainability | | |
|---|---|---|---|---|
| | | Before | After | Change |
| argouml | Main | 72.3667 | 77.1000 | -4.7333 |
| argouml | ProjectBrowser* | 194.4034 | 194.6701 | -0.2667 |
| argouml | Configuration* | 52.2333 | 52.5000 | -0.2667 |
| argouml | ActionSaveGIF* | 27.9387 | 27.9358 | 0.0029 |
| argouml | ActionSaveGraphics | 34.2556 | 38.1477 | -3.8921 |

| | | | | |
|---|---|---|---|---|
| argouml | BuildException* | 287.4333 | 288.5000 | -1.0667 |
| ant | Project* | 464.5000 | 465.0333 | -0.5333 |
| ant | ProjectComponent* | 237.9667 | 239.8333 | -1.8667 |
| ant | Task* | 208.7000 | 210.8333 | -2.1333 |
| ant | UnknownElement* | 148.7667 | 149.0333 | -0.2667 |
| ant | TarEntry* | 198.4667 | 198.7333 | -0.2667 |
| ant | Copy | 250.3772 | 253.8771 | -3.4999 |
| ant | Delete | 166.7000 | 169.0333 | -2.3333 |
| ant | ExecTask* | 133.0333 | 133.0388 | -0.0055 |
| ant | Execute* | 148.5667 | 148.8333 | -0.2667 |
| ant | ExecuteJava | 73.5333 | 75.1333 | -1.6000 |
| ant | MacroDef* | 83.2000 | 83.4667 | -0.2667 |
| ant | MacroInstance | 104.8667 | 107.6333 | -2.7667 |
| ant | Parallel | 73.5333 | 76.4000 | -2.8667 |
| ant | Tar | 159.2753 | 163.3752 | -4.0999 |
| ant | XSLTProcess | 208.0333 | 209.8667 | -1.8333 |
| ant | Zip | 419.4352 | 424.5351 | -5.1000 |
| ant | AbstractFileSet* | 221.0493 | 221.3159 | -0.2667 |
| ant | ArchiveFileSet* | 115.9667 | 116.5000 | -0.5333 |
| ant | CommandlineJava* | 149.0000 | 149.8000 | -0.8000 |
| ant | Resource* | 131.5000 | 132.0333 | -0.5333 |
| ant | Watchdog* | 18.5333 | 18.8000 | -0.2667 |
| ant | Tar$TarFileSet* | 11.6333 | 11.9000 | -0.2667 |
| ant | Commandline$Argument* | 65.0667 | 65.6000 | -0.5333 |
| ant | BaseTest* | 36.9667 | 37.2333 | -0.2667 |
| ant | JUnitTask | 419.9075 | 427.1087 | -7.2012 |

| | | | | |
|---|---|---|---|---|
| ant | JUnitTest | 102.9667 | 105.4667 | -2.5000 |
| ant | JUnitTestRunner | 277.9667 | 281.6667 | -3.7000 |
| log4j | FileAppender* | 46.4223 | 46.6890 | -0.2667 |
| log4j | CustomSQLDBReceiver* | 33.3728 | 33.3725 | 0.0003 |
| log4j | DBAppender* | 63.6333 | 67.2333 | -3.6000 |
| log4j | DBHelper* | 12.0667 | 12.3333 | -0.2667 |
| log4j | ScanError* | 6.7667 | 7.0333 | -0.2667 |
| log4j | Token* | 23.8000 | 24.8667 | -1.0667 |
| log4j | TokenStream | 48.8667 | 54.5500 | -5.6833 |
| log4j | RollingFileAppender | 70.9452 | 70.2295 | 0.7157 |
| log4j | ComponentBase* | 60.8667 | 61.1333 | -0.2667 |
| log4j | LoggingEvent* | 165.2333 | 165.7667 | -0.5333 |
| log4j | LogFilePatternReceiver | 138.9354 | 143.6020 | -4.6666 |
| log4j | CustomSQLDBReceiver$CustomReceiverJob | 46.3333 | 53.9667 | -7.6333 |
| xerces | XMLErrorReporter* | 97.51795 | 98.05128 | -0.5333 |
| xerces | XML11NSDTDValidator | 46.3849 | 47.97495 | -1.5901 |
| xerces | XMLNSDTDValidator | 47.5454 | 49.01054 | -1.4651 |
| xerces | XMLSchemaFactory | 96.4602 | 98.49325 | -2.0331 |
| xerces | XSCMLeaf* | 19.03333 | 19.3 | -0.2667 |
| xerces | XSDFACM | 143.581 | 145.0782 | -1.4972 |

**Observations:** as shown in Table 11, 22 instances of large method bad smells were refactored on the five systems under study. We applied the refactoring technique Extract Method.to these classes. We found out that all classes had a decrease in their maintainability after removing the bad smell, except for one class; RollingFileAppender. After investigating this class, we found that

its maintainability improved because the refactoring process extracted a duplicated code from two places in the method. This improvement in duplication led to an improvement in maintainability.

For the classes indirectly affected by the refactoring process, and are denoted by a * in the table, the maintainability decreased in all classes but two class, ActionSaveGIF and CustomSQLDBReceiver. After investigating these two classes, we found out that the improvement is attributed to a change in duplication values in the classes. For example, we found that a small clone instance was present in ActionSaveGIF and another refactored class. After refactoring, this clone instance was modified in the refactored class which lead to a decrease in the duplication ratio of ActionSaveGIF (i.e. its CC value decreases). This is the only change to the metrics values of the class, which led to an increase in its maintainability. Based on this, the null hypothesis $H3_0$ is rejected and the alternative hypothesis $H3_a$ is accepted.

### 5.7.4 Large Class

Large Class smell instances detected and reported by the tool in each system are presented in Table 13, and the number of instances that were verified to be bad smells and refactored. Table 14 list the classes affected by the refactoring of Large Class smell and their maintainability value change.

**Table 13: Summary for Large Class Audit.**

| System | Detected Bad Smell Possibilities | Refactored Bad smells instances |
|---|---|---|
| Ant | 4 | 3 |
| ArgoUML | 0 | 0 |
| Log4j | 0 | 0 |
| JHotDraw | 2 | 0 |
| Xerces-j | 1 | 0 |
| **Total** | **7** | **3** |

**Table 14: Comparison of the maintainability of affected classes before and after Large Class.**

| System | Class | Maintainability | | |
|---|---|---|---|---|
| | | Before | After | Change |
| ant | Tar | 159.27525 | 159.27556 | -0.00030 |
| ant | Zip | 419.43516 | 419.43519 | -0.00003 |
| ant | JUnitTask | 419.90754 | 419.90763 | -0.00009 |

**Observations:** Three instances of Large Class were refactored in the systems under study. The refactoring and removal of large class bad smells, by applying Extract Class refactoring technique, appears to have a negative impact on the class maintainability. All refactored classes show a decrease on their maintainability value after refactoring. Based on this, we can reject the null hypothesis $H4_0$ and accept the alternative hypothesis $H4_a$.

### 5.7.5 Data Class

Table 15 presents the number of bad smells instances detected and reported by the tool in each system, and the number of instances that were verified to be bad smells and, in turn, refactored. Also, Table 16 list the classes affected by the refactoring of Data Class instances and their maintainability value change.

**Table 15: Summary for Data Class Audit.**

| **System** | **Detected Bad Smell Possibilities** | **Refactored Bad smells instances** |
|---|---|---|
| Ant | 2 | 2 |
| ArgoUML | 9 | 6 |
| Log4j | 0 | 0 |

| | | |
|---|---|---|
| JHotDraw | 0 | 0 |
| Xerces-j | 11 | 7 |
| **Total** | **22** | **15** |

**Table 16: Comparison of the maintainability of affected classes before and after Data Class**

| System | Class | Maintainability | | |
|---|---|---|---|---|
| | | Before | After | Change |
| ant | BuildEvent | 33.73333 | 33.76667 | -0.03333 |
| ant | Project* | 464.50000 | 462.16667 | 2.33333 |
| ant | BorlandDeploymentTool* | 96.27165 | 96.27164 | 0.00001 |
| ant | EjbJar* | 68.50622 | 68.50513 | 0.00109 |
| ant | JonasDeploymentTool* | 136.74827 | 136.74822 | 0.00004 |
| ant | WeblogicDeploymentTool* | 154.04139 | 154.04133 | 0.00006 |
| ant | WeblogicTOPLinkDeploymentTool* | 14.87421 | 14.87488 | -0.00067 |
| ant | WebsphereDeploymentTool* | 153.44914 | 153.44912 | 0.00001 |
| ant | EjbJar$Config | 2.16667 | 40.40000 | -38.23333 |
| argouml | HistoryItem | 16.83333 | 18.90000 | -2.06667 |
| argouml | Actions | 10.13333 | 38.20000 | -28.06667 |
| argouml | ProjectBrowser* | 194.40342 | 194.40340 | 0.00001 |
| argouml | InitMenusLater* | 31.99041 | 31.99071 | -0.00030 |
| argouml | ToDoTreeRenderer | 22.03333 | 23.40000 | -1.36667 |
| argouml | ActionAggregation | 10.54148 | 17.79746 | -7.25598 |
| argouml | ActionCompartmentDisplay | 13.80000 | 21.06667 | -7.26667 |
| argouml | ActionMultiplicity | 10.86871 | 20.45706 | -9.58835 |
| argouml | FigAssociation* | 59.62667 | 59.62762 | -0.00095 |
| argouml | FigClass* | 131.56185 | 131.56176 | 0.00009 |

| | | | | |
|---|---|---|---|---|
| xerces | NodeImpl | 16.46667 | 29.73333 | -13.26666 |
| xerces | ElementState | 2.333333 | 38.23333 | -35.90000 |
| xerces | DOMLocatorImpl | 20.83333 | 30.3 | -9.46667 |
| xerces | DOMErrorImpl | 17.9 | 28.96667 | -11.06667 |
| xerces | XMLSimpleType | 22.33333 | 43.66667 | -21.33334 |
| xerces | XMLEntityDecl | 13.73333 | 28.6 | -14.86667 |
| xerces | HTMLSerializer | 198.3212 | 198.3213 | -0.00010 |
| xerces | TextSerializer* | 70.32044 | 70.32082 | -0.00038 |
| xerces | XML11Serializer* | 122.2373 | 122.2375 | -0.00020 |
| xerces | XMLSerializer* | 338.2569 | 338.2573 | -0.00040 |
| xerces | EventImpl* | 18.5 | 23.56667 | -5.06667 |
| xerces | DOMErrorHandlerWrapper* | 47.35266 | 47.35283 | -0.00017 |
| xerces | BaseMarkupSerializer* | 431.5225 | 431.5225 | -0.00005 |
| xerces | DocumentImpl* | 256.1476 | 256.1477 | -0.00010 |

**Observations:** 15 instances of data class bad smells were refactored in the studied systems. We used the refactoring method Encapsulate Field for 14 of these classes, and we used Remove Setting Method for one class. After refactoring, we found out the removal of data class bad smell affected the class's maintainability in a negative way; i.e. resulted in a decrease in maintainability value.

For the classes indirectly affected by the refactoring process, which are denoted by a * in the table above, we have two distinct observations. First, the maintainability change was very minimal and close to zero in all classes that use a class refactored by Encapsulate Field refactor method. In addition to that, the change is always attributed to a minimal change in CC values. On the other hand, for the one class that uses the class refactored by Remove Setting Method, the maintainability had a clear improvement. This further supports our claim that refactoring bad smell

has different impact depending on the refactoring method used. Based on this data, the null hypothesis $H5_0$ is rejected, and the alternative hypothesis $H5_a$ is accepted.

## 5.8 Discussion

Based on the result of the previous sections, we can say that the impact of bad smell removal on class maintainability varied from one bad smell to another. Also, it may varies depending on the type of class under consideration and whether it was refactored rather than being affected by the refactoring process in an indirect way.

To summarize the finding of our experiment, we present Table 17 where we represented the effect of each bad smell on each type of class affected by the refactoring process. We have two type of classes:

- Refactored classes:

  The classes that are flagged by the audit process as classes that needs to be refactored, and classes which were directly modified in the refactoring process to get rid of the bad smell. We used one or more of the refactoring methods to improve the maintainability of these classes.

- Affected classes:

  The classes that were changed in terms of their maintainability values after the refactoring process was performed on the refactored classes. These classes are usually using, calling, or modifying the refactored classes, and because the refactored classes were change, some

changes to their underlying metric values resulted in the change of their maintainability values.

For each system, we looked at each type of classes with each one of the bad smell considered in this study. If we see that all classes of that type and with that smell has improved in terms of quality, we make the corresponding\ng cell with a ↑. We marked the cell with a ↓ if all class has shown a decrease in their maintainability values. If no clear change is observed, we make the cell with --. N/A mean that there are no classes of that specific system with this type and bad smell and refactoring method.

**Table 17: Summary of Experiment Findings.**

| | **Ant** | | **ArgoUML** | | **Log4j** | | **JHotDraw** | | **Xerces-J** | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Bad Smell & Refactoring Method** | **R.C.** | **A.C.** | **R.C.** | **A.C.** | **R.C.** | **A.C.** | **R.C.** | **A.C.** | **R.C.** | **A.C.** |
| Feature Envy with Extract Method w/o duplication | ↓ | N/A | ↓ | N/A | ↓ | ↓ | ↓ | N/A | N/A | N/A |
| Feature Envy with Extract Method w/ duplication | N/A | N/A | ↑ | N/A | N/A | N/A | ↑ | N/A | N/A | N/A |
| Feature Envy with Move Method from class | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | ↑ | N/A |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Feature Envy with Move Method to class | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | ↓ | N/A |
| Shotgun Surgery with Move Fields to Class | N/A | N/A | ↓ | ↓ | ↓ | N/A | N/A | N/A | N/A | N/A |
| Shotgun Surgery with Move Fields from Class | ↑ | N/A | ↑ | N/A | ↑ | N/A | N/A | N/A | N/A | N/A |
| Large Method with Extract Method w/o duplication | ↓ | N/A | ↓ | ↓ | ↓ | N/A | N/A | N/A | ↓ | ↓ |
| Large Method with Extract Method w/ duplication | N/A | N/A | N/A | N/A | ↑ | N/A | N/A | N/A | N/A | N/A |
| Large Class with Extract Class | ↓ | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| Data Class with Encapsulate Field | ↓ | -- | ↓ | -- | N/A | N/A | N/A | N/A | ↓ | ↓ |
| Data Class with Remove Setting Method | ↓ | ↑ | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |

The above table present a clear trend of the impact of bad smell on refactored classes for each combination of bad smell type and refactoring method. To quantify this impact, for each combination of bad smell and refactoring method in our experiment, we will average the maintainability change values of the classes. This will give us an idea of how much each bad smell refactoring is affecting the maintainability of the refactored class.

Table 18 presents each bad smell and refactoring method along with their average impact on maintainability; i.e. the change of maintainability value of the classes after refactoring. The second column represent the number of classes that fall under this type of bad smell and refactoring method.

**Table 18: List of Bad Smells and Refactoring methods and their average Impact on maintainability**

| **Bad Smell & Refactoring Method** | **# of Instances** | **Avg. Impact** |
|---|---|---|
| Feature Envy: Extracted Method from class without duplication | 4 | -2.058157167 |
| Feature Envy: Extracted Method from class with duplication | 2 | 1.795435267 |
| Feature Envy: Move Method from class | 1 | 4.266666667 |
| Feature Envy: Move Method to class | 1 | -4.533333333 |
| Shutgun Surgery: Move Fieled to class | 4 | -3.741666667 |
| Shutgun Surgery: Move Fieled from class | 3 | 15 |
| Large Class: Extract Class | 3 | -0.00013912 |
| Data Class: Encapsulate Field | 14 | -14.26746153 |
| Data Class: Remove Setting Method | 1 | -0.033333333 |
| Large Method: Extract Method without duplication | 21 | -3.220021813 |
| Large Method: Extract Method w/ duplication | 1 | 0.715689733 |

# CHAPTER 6

# Bad Smells Prioritization Model (BSPM)

This chapter describes the proposed prioritization framework for bad smell and its application based on the experiment conducted in Chapter 6. The first section will describe the model and the different steps in details. The second section will apply the model to the result obtained in the previous chapter. The third section will present expert evaluation of the proposed model, while the fourth section will explore the relationship between bad smells ranking and its maintainability impact.

## 6.1 The Proposed Approach for Bad Smell Prioritization

Our proposed model will assist software managers in ordering bad smells based on their impact on maintainability. We think knowing the impact of bad smells on the maintainability, as discussed earlier, is an important step. It helps in understanding the importance of bad smells and their impact on software quality. However, we believe that knowing the impact is not enough. We think that having an order list of bad smells based on their impact quality is of great value to practitioners.

In our model, we utilized the Analytical Hierarchy Process [56] which will help in prioritizing the bad smells based on their maintainability impact. The outline of our model is presented in Figure 9. The following steps explain how one might use the model:

1. Elicit Bad Smells Candidate List: In this step, we select the bad smells we want to investigate and prioritize.

2. Calculate Impact: In this step, the impact of bad smell is calculated. This is done by utilizing the maturity model proposed earlier to measure the maintainability change before and after refactoring. The result of this stage is list of bad smells and refactoring combinations and their average impact on maintainability.

3. Scale Impact: In this step, we take the bad smell impact, which is the output of the previous step, and scale it to a positive scale. This will enable us to use AHP in our model while preserving the difference of impact between different bad smells. The output of this step is the list pf bad smells with their adjusted impact.

4. Apply AHP pairwise comparison: In this step, we generate a matrix to represent the relative impact of one bad smell to another. In this matrix, for each cell $a_{ij}$,its value will be the adjusted impact of bad smell $BS_i$ divided by the adjusted impact of bad smell $BS_j$. We do this for each pair of bad smells under consideration.

5. Calculate weights: The final step is calculating the weight for each bad smell. This is done by normalizing any column of the matrix, as suggested by Harker [57]. The simple normalization is possible if the matrix is perfectly consistent, i.e.:

$$a_{ik} * a_{kj} = a_{ij} \quad \text{for all I, j, k = 1, 2, ...., n,}$$

which is always true in our proposed model. The result of this step is a list of bad smells and their relative impact on maintainability. Normalizing the values will help in presenting the impact in a clean way.

As mentioned earlier, we utilized a part of AHP's technique in our model. AHP is a method to analyze and find the best option from a set of possible options. First, AHP's users will break down the decision into a hierarchy of criteria and alternatives. This process is called decision modeling. Then, for each criterion, users of AHP will evaluate the importance of the criterion by comparing every pair of alternative. This is done by filling a matrix with numerical values where the value of $a_{ij}$ in the matrix denotes how much alternative i is important with respect to alternative j. The value can range from 1 to 9, where 1 means Equally important and 9 means Extremely important. Then after evaluating all the criteria, users can derive priorities for the alternative options. The option with the highest priority is the best option.

In our model, we utilize the pairwise comparison to compare every pair of bad smells under study (i.e. the alternatives) with respect to their effect on maintainability (i.e. criteria). There is consistency check process in AHP which is useful to detect any inconsistency in the judgment of the users. Inconsistency can happen when a user assigns a value of 2 to option A with respect to option B, and assign 2 to option B with respect to C, but assigns 3 (and not 4) to option A with respect to C. However, in our model, we do not use subjective values.

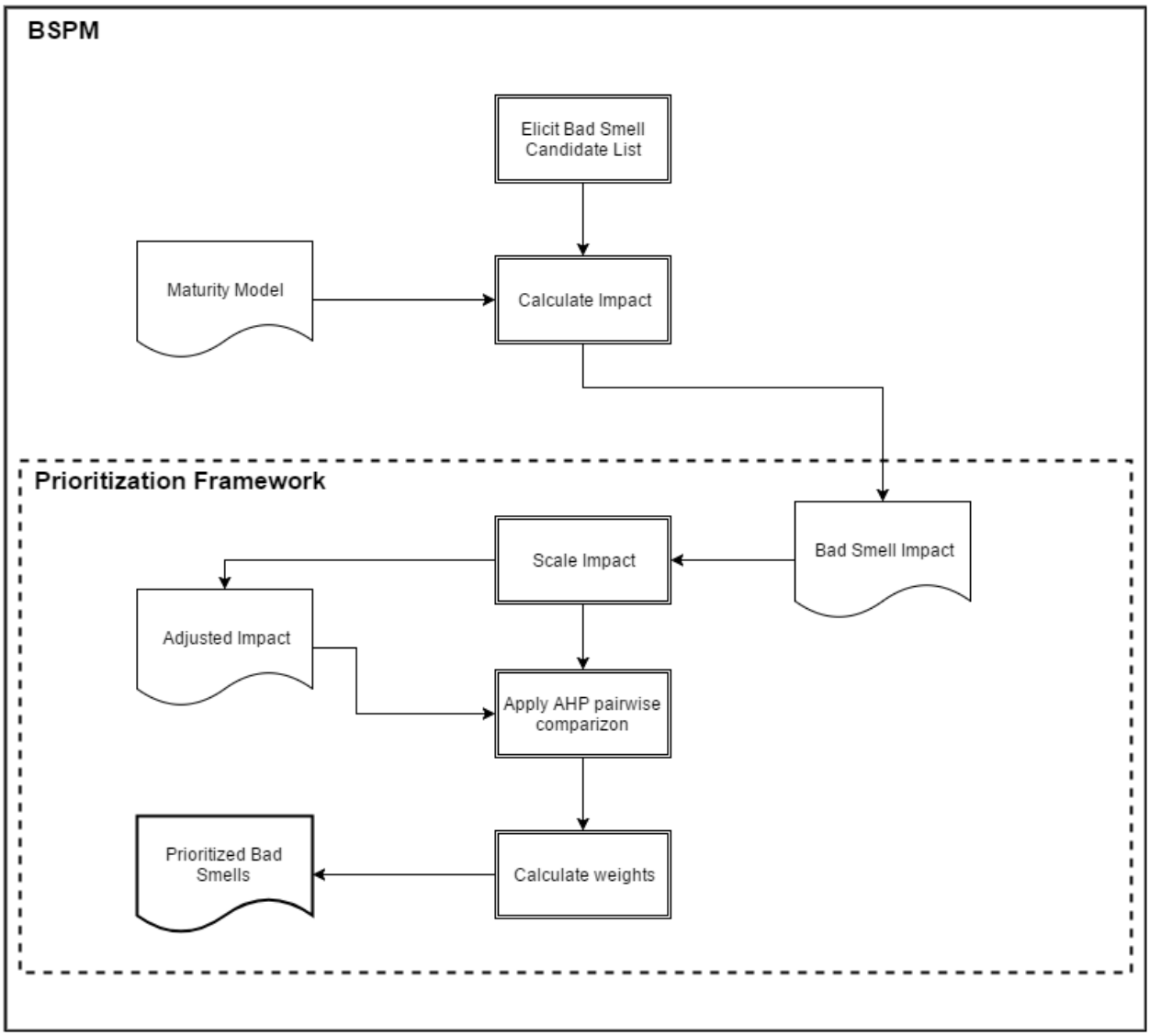



**Figure 9: The proposed Bad Smell Prioritization Model (BSPM)**

## 6.2 Applying BSPM

In this section, we will use the data from our experiment to prioritize bad smells based on their impact on maintainability. We treated every combination of bad smell and refactoring method

applied as a separate case that needs to be prioritized. Table 19 presents the bad smells considered in our prioritization along with their average impact on maintainability.

**Table 19: Bad Smells Considered for Prioritization.**

| Bad Smell & Refactoring Method | Avg. Impact |
|---|---|
| Feature Envy: Extracted Method from class without duplication | -2.058157167 |
| Feature Envy: Extracted Method from class with duplication | 1.795435267 |
| Feature Envy: Move Method from class | 4.266666667 |
| Feature Envy: Move Method to class | -4.533333333 |
| Shutgun Surgery: Move Fieled to class | -3.741666667 |
| Shutgun Surgery: Move Fieled from class | 15 |
| Large Class: Extract Class | -0.00013912 |
| Data Class: Encapsulate Field | -14.26746153 |
| Data Class: Remove Setting Method | -0.033333333 |
| Large Method: Extract Method without duplication | -3.220021813 |
| Large Method: Extract Method w/ duplication | 0.715689733 |

The next step would be to scale the impact to a positive scale. As discussed earlier, this will allow us to utilize AHP in our prioritization process. Based on previous experience, it was recommended that 9 is a good maximum value to use [58]. Table 20 is showing the adjusted impact for each bad smell in our prioritization process. As you can see in the table, the values are ranging from 1 to 9.

**Table 20: Bad Smell Adjusted Impact.**

| Bad Smell & Refactoring Method | Adj. Impact |
|---|---|
| Feature Envy: Extracted Method from class w/o duplication | 4.337304632 |
| Feature Envy: Extracted Method from class w/ duplication | 5.390649809 |
| Feature Envy: Move Method from class | 6.066138908 |
| Feature Envy: Move Method to class | 3.660737267 |
| Shutgun Surgery: Move Fieled to class | 3.877132301 |
| Shutgun Surgery: Move Fieled from class | 9 |
| Large Class: Extract Class | 4.899845539 |
| Data Class: Encapsulate Field | 1 |

| | |
|---|---|
| Data Class: Remove Setting Method | 4.890772197 |
| Large Method: Extract Method w/o duplication | 4.019719276 |
| Large Method: Extract Method w/ duplication | 5.095510982 |

The next step would be to apply AHP's pairwise comparison to each pair of bad smells. This will result in a matrix where each cell represents the relative impact of one bad smell on maintainability compared to another bad smell impact. The resulting matrix is presented in Table 21.

**Table 21: Pairwise Comparison Matrix for Bad Smells**

| | FE: EM w/o duplication | FE: EM w/ duplication | FE: MM from class | FE: MM to class | SS: MF to class | SS: MF from class | LC: EC | DC: EF | DC: RSM | LM: EM w/o duplication | LM: EM w/ duplication |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FE: EM w/o duplication | 1 | 0.804597736 | 0.715002524 | 1.184817242 | 1.118688839 | 0.481922737 | 0.885192114 | 4.337304632 | 0.886834319 | 1.079006849 | 0.851201115 |
| FE: EM w/ duplication | 1.242857089 | 1 | 0.888645956 | 1.472558508 | 1.390370354 | 0.59896109 | 1.100167294 | 5.390649809 | 1.102208321 | 1.341051312 | 1.057921341 |
| FE: MM from class | 1.398596461 | 1.125307546 | 1 | 1.657081201 | 1.564594251 | 0.674015434 | 1.238026558 | 6.066138908 | 1.24032334 | 1.509095161 | 1.190486868 |
| FE: MM to class | 0.844012025 | 0.679090165 | 0.603470729 | 1 | 0.944186833 | 0.406748585 | 0.747112789 | 3.660737267 | 0.74849883 | 0.910694756 | 0.718423977 |
| SS: MF to class | 0.893903618 | 0.719232827 | 0.639143343 | 1.059112419 | 1 | 0.430792478 | 0.791276433 | 3.877132301 | 0.792744406 | 0.964528126 | 0.760891756 |
| SS: MF from class | 2.075021416 | 1.669557534 | 1.483645551 | 2.458521151 | 2.321303299 | 1 | 1.836792594 | 9 | 1.840200205 | 2.238962321 | 1.766260544 |
| LC: EC | 1.129696827 | 0.908952267 | 0.807737115 | 1.338485989 | 1.263780846 | 0.544427282 | 1 | 4.899845539 | 1.001855196 | 1.218952171 | 0.961600428 |
| DC: EF | 0.230557935 | 0.185506393 | 0.164849506 | 0.273169017 | 0.257922589 | 0.111111111 | 0.204088066 | 1 | 0.204466689 | 0.248773591 | 0.196251172 |
| DC: RSM | 1.127606339 | 0.907269507 | 0.806241379 | 1.336007432 | 1.261440626 | 0.543419133 | 0.998148239 | 4.890772197 | 1 | 1.216694963 | 0.959819773 |
| LM: EM w/o duplication | 0.926778176 | 0.745683622 | 0.662648735 | 1.098062763 | 1.036776402 | 0.446635475 | 0.820376733 | 4.019719276 | 0.821898693 | 1 | 0.788874617 |
| LM: EM w/ duplication | 1.174810491 | 0.945249861 | 0.839992466 | 1.391935725 | 1.314247384 | 0.566167887 | 1.039932982 | 5.095510982 | 1.041862262 | 1.267628566 | 1 |

*Abbreviation used in the table*: FE: Feature Envy, SS: Shotgun Surgery, LC: Large Class, DC: Data Class, LM: Large Method, EM: Extract Method, MM: Move Method, MF: Move Field, EC: Extract Class, EF: Extract Field, RSM: Remove Setting Method.

The final step is to calculate the weight for each bad smell and order the list of bad smells based on their impact on maintainability. This is done by normalizing any column in the pairwise comparison matrix to come up with the final relative impact of each bad smell. The result of this normalization is presented in Table 22.

**Table 22: Relative Weighted Impact of Bad Smells**

| **Bad Smell & Refactoring Method** | **Relative weighted Impact** |
|---|---|
| Feature Envy: Extracted Method from class w/o duplication | 0.083029985 |
| Feature Envy: Extracted Method from class w/ duplication | 0.103194405 |
| Feature Envy: Move Method from class | 0.116125443 |
| Feature Envy: Move Method to class | 0.070078305 |
| Shutgun Surgery: Move Fieled to class | 0.074220804 |
| Shutgun Surgery: Move Fieled from class | 0.172288996 |
| Large Class: Extract Class | 0.09379883 |
| Data Class: Encapsulate Field | 0.019143222 |
| Data Class: Remove Setting Method | 0.093625137 |
| Large Method: Extract Method w/o duplication | 0.076950378 |
| Large Method: Extract Method w/ duplication | 0.097544497 |
| **Total** | **1** |

Based on the presented result, we can rank bad smells based on their relative impact on maintainability compared to other bad smells. The ranking is shown in *Table 23*.

**Table 23: Bad Smells Ranking**

| **Bad Smell & Refactoring Method** | **Ranking** |
|---|---|
| Shutgun Surgery: Move Fieled from class | 1 |
| Feature Envy: Move Method from class | 2 |
| Feature Envy: Extracted Method from class w/ duplication | 3 |
| Large Method: Extract Method w/ duplication | 4 |
| Large Class: Extract Class | 5 |
| Data Class: Remove Setting Method | 6 |
| Feature Envy: Extracted Method from class w/o duplication | 7 |
| Large Method: Extract Method w/o duplication | 8 |

| | |
|---|---|
| Shutgun Surgery: Move Fieled to class | 9 |
| Feature Envy: Move Method to class | 10 |
| Data Class: Encapsulate Field | 11 |

## 6.3 Expert Evaluation of BSPM

Although maturity models are presented as an evaluation instrument, maturity models should be subjected to evaluation and assessment. The evaluation process should aim to understand the maturity model and improve it further. When it comes to maturity model evaluations, there are three variations [59]: (i) **Author-based:** An evaluation conducted by the model's authors who will evaluate and compare their proposed model to other models, (ii) **Practical-based:** Where the maturity model is used in a practical setting, and (iii) **Expert-based:** where the maturity model is evaluated by domain experts who were not involved in the development of the model. This is usually done through surveys and interviews.

We presented our proposed model to five experts and asked them to fill a short survey to assess the model. Their backgrounds cover both industry and academia. The evaluation form is presented in Appendix 2. The following table presents the summary of the responses:

Table 24: Summary of the Experts evaluation of the BSPM.

| **Question** | **Median Response (N=5) (1= Strongly Disagree, 5= Strongly Agree)** |
|---|---|
| The BSPM model is simple | 4 |
| The BSPM model is easy to understand | 4 |
| The BSPM model is easy to use | 5 |

| | |
|---|---|
| The BSPM model is flexible | 4 |
| The BSPM model is practical for use in industry | 4 |
| The BSPM model is a useful tool for practitioners | 4 |

Most experts agree that the prioritization model presented here is simple, flexible, and easy to understand, with median response score of 4. They all agree that the model is easy to use. When asked about the practicality and usefulness of the model in the industry, they mostly agree that the model is practical and useful in professional settings.

Maturity models should possess certain qualities in order for it to be of value. Two of the basic qualities of any model are the simplicity and understandability of the model. In order for the model to be used and employed in professional settings, one might strive to make the model easier to use and of certain desired value and usefulness to its users. One of the improvements that were suggested by the evaluators is to automate the model as possible. Such automation will save time and make the model easier to use and less prone to human errors.

Another main suggestion that echoed across was to incorporate other factors in our model that might affect the developers' decision when it comes to choosing a bad smell to factor. This suggestion hints that the benefit of bad smell refactoring (in terms of improvement to software quality) might not be the only concern of the maintainer. Other concerns might be the time and/or cost needed to refactor the bad smell, so having known the benefit of removing a bad smell and its cost might make the model more practical and useful for practitioners.

## 6.4 The Relationship Between Bad Smell Ranking and its Impact on Maintainability.

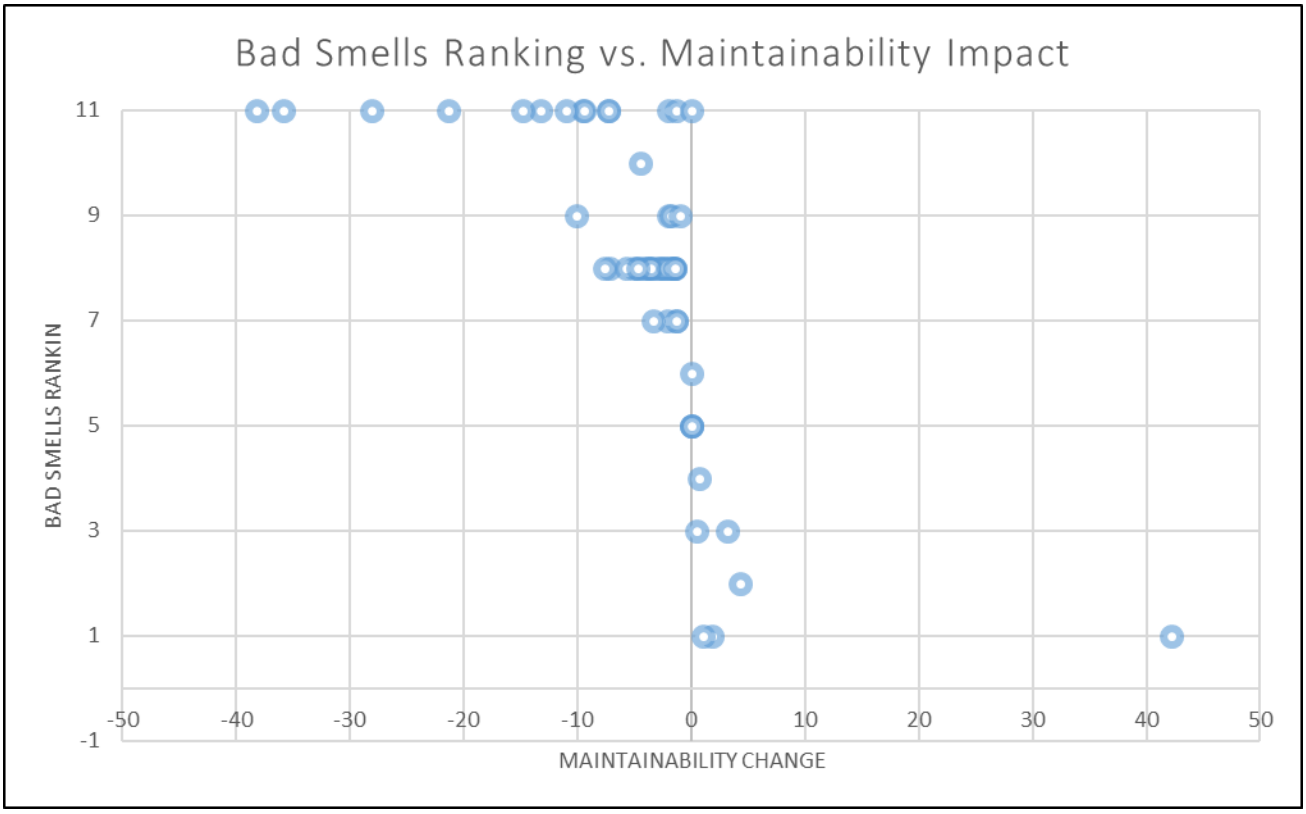



**Figure 10: Bad Smells Ranking compared to maintainability change for each class.**

Figure 10 shows the relathionship between bad smells ranking and its impact on maintainability. Each circle represents a refactored class in our experiement, where the x-axies represents the change in the class's maintainability, and the y-axis represents the ranking of the bad smell in the class as concluded in the previous section. The ranking goes from 1 to 11, i.e.: a bad smell that is ranked 1 is more important than a bad smell ranked with 11. APPENDIX I MAINTAINABILITY CHANGE & BAD SMELLS RANKING shows the maintainability change and bad smell ranking for each refactored class instance in our experiement. A negative

maintainability change for a class means that the cost of maintaining the class has increased after refactoring the class for that particular bad smell, i.e. the class becomes much harder to maintain. On the other hand, a positive maintainability change means the class's maintainability cost have decreased after refactoring, i.e. the refactored class is easier to maintain.

As the above graph shows, as the bad smell ranking go down, the maintainability of the class decreases. For example, refactoring for Shutgun Surgery by moving the fields from the class (which is ranked #1 in our model) will generally have a positive change in maintainability. However, when we look at the process of refactoring a Data Class bad smell by using Encapsulate Field refactoring method (which is ranked #11) we can see a clear negative impact on the class's maintainability. This is what we expect from our proposed model, to rank bad smells from top to bottom based on their relative impact on the maintainability of the classes after performing the refactoring process.

## 6.5 Customization of BSPM

The Bad Smell Prioritization Model was designed to quantify the effect of bad smell on software quality, and prioritize the smells based on that effect. In our case, we think of maintainability as a measure for the software quality. However, our model is highly adaptable to other software qualities. This section discusses the process of adapting this model to other quality attributes.

First, you need to specify the quality attribute you want to use and its sub-characteristics if any. It is highly recommended to use a well-defined popular standard to define your quality attribute. In our model, we used the up-to-date ISO 25010's definition of maintainability and its sub-characteristics [49].

Next, you need to use a maturity model for your quality attribute. You may not find a suitable model for your quality attribute or you may find one that does not fit your objectives exactly. This may lead you to modify an existing model to fit your needs, or develop a new maturity model from scratch. In our proposed model, we used a modified version of the SIG model [48].

By doing the above, you will be able to adapt our Bad Smell Prioritization Model to your quality attributes and using your proposed maturity model. One advice we can give you is to use a well-known model that have been proven mathematically and have been used a lot and applied in practice. This will give you confidence in the results produced by your model.

# CHAPTER 7

# CONCLUSION & FUTURE WORK

Tackling bad smells is a great way to improve the quality of the software. They are in most cases a good indicator of code problems. Refactoring them should be considered in the process of maintaining the software. Recent work has considered the effectiveness of refactoring bad smells in general, which was insightful in the process of understanding bad smells and their effect on the overall quality of the software. However, there was a need to explore and study the impact of bad smells in software quality compared to one another. Knowing which flaw has greater impact than the other will help software professionals to better utilize their limited resources.

We proposed a bad smell prioritization model in this research. Our approach to prioritizing starts with system audit to know where the weakness in the code is. This is done by analyzing the system looking for bad smell instances. Our next step is to analyze each bad smell instances and see if the bad smell points to a genuine problem or not. The next step is to evaluate the system before applying the refactoring process. Next, we refactored the system in iteration, each iteration is for one bad smell type detected in the system. We used a set of refactoring methods which are known to be a possible solution to each refactoring. After that, we evaluate the result code of the iterations once again. Finally, we compare the effect the bad smell had on the system after its removal.

Such prioritization models will help greatly in reducing the cost and time of the maintenance process whilst improving the overall quality of the software. We believe that the proposed model will help software professionals analyze and address the most effective bad smells in terms of their

impact on quality. We think that more research is needed to study the effect of bad smells on software quality with respect to each other.

## 7.1 Contributions

The research work presented in this thesis makes the following contribution to the field of bad-smell impact on quality:

- Presents an in-depth survey and review in the field of bad smell impact on different quality attributes and maturity models.
- Evaluated the impact of five bad smells on software maintainability.
- Provides A prioritization approach of bad smells based on how much they affect software maintainability.
- Provides a visualized view of the correlation between bad smells ranking and their impact on maintainability.

## 7.2 Threats to Validity

**Construct Validity**

The metrics used to measure the software properties might not be the perfect fit. However, we relied on the metrics selected by the authors of the original maturity model. After tweaking the model to fit our research objectives, we used the metric used in the original model when possible. To confirm to our research context, we used well-defined well-known metrics to measure the properties.

Another threat to validity is the choice of threshold to identify the bad smell instances. The choice of threshold values will have a clear impact on the accuracy of the audit process and in identifying bad smell instances.

Another threat to validity is the refactoring process performed by the researcher. As the tool used to identify bad smell is using thresholds to do so, a manual inspection and verification of bad smell instances correctness is needed. Then the refactoring process is done in a semi-automatic way, with the assistance and suggestions provided by the tool.

In our proposed model, we used the SIG model and the AHP approach, and assumed that they are sound and correct. Although SIG model was built by expert in the field of evaluating maintainability, several empirical studies have been performed to assert the soundness of the model [60, 61]. The findings of these studies were mostly consistent with expert opinion and the ratings produced by the model. For the AHP approach, it is worth pointing out that the approach have been used widely in practice [62]. In addition to its wide use and applications, the approach's theoretical basis has been proven and validated [58, 63].

**Internal validity**

One threat to validity is the refactoring of bad smells may influence other bad smells; this effect may be positive or negative one. The refactoring may also introduce new bad smells to the code post refactoring. We do refactoring for each smell in an independence from other smells, i.e. for each smell to refactor, we take the original un-refactored code. When we move to the next smell, we start again from the original code. If one bad smell is affecting or introducing other bad smells, we can think of this as side effects of refactoring that particular type of smell.

**External validity**

The systems under study are Java-based open source systems. They are maintained by an online community, and this may hinder the ability to generalize the results of the experiment to professional setting and to other programming languages.

Our research focused on five bad smells. There might be other bad smells that have different impact on the software.

## 7.3 Future Work

There is a clear lack of maturity models specific to the process of bad smell detection and refactoring. The work developed in this research presented a proposed framework to evaluate and prioritized bad smells. One area of future improvement is to evaluate the approach using real-world projects and applications, with different domains and sizes. Such evaluation will help in improving the model and evaluating its effectiveness.

Other paths for future work include the expansion of the approach to cover more bad smells. The presented here discussed a subset of bad smells and evaluated them using the proposed model. Future work should expand on that and explore the possibilities of applying the approach on a wider selection of bad smells.

In the future, an automated tool or plugin can be developed to ease the use of the model. Such automation will have great impact on the wide use of the approach. An automated process will ensure the minimization of the evaluation process. It will also streamline the process of evaluation and saves precious time in the maintenance process.

# APPENDIX Ⅰ

# MAINTAINABILITY CHANGE & BAD SMELLS RANKING

*Table 25: Maintainability change and ranking of bad smell for each refactored class instance*

| Class | Maintainability Change | Bad Smell Ranking |
|---|---|---|
| DOMUtil | -1.366666667 | 7 |
| ActionAddNote | -2.1 | 7 |
| PropPanel | 0.433333333 | 3 |
| LBELEventEvaluator | -3.4 | 7 |
| JUnitTest | 42.16666667 | 1 |
| ArgoEvent | -2 | 9 |
| Critic | 1.833333333 | 1 |
| KnowledgeTypeNode | -1.833333333 | 9 |
| UMLAction | -10.13333333 | 9 |
| Util | -1 | 9 |
| ConnectionSource | 1 | 1 |
| Main | -4.733333333 | 8 |
| ActionSaveGraphics | -3.892082933 | 8 |
| Copy | -3.4999358 | 8 |
| Delete | -2.333333333 | 8 |
| ExecuteJava | -1.6 | 8 |
| MacroInstance | -2.766666667 | 8 |
| Parallel | -2.866666667 | 8 |
| Tar | -4.0999088 | 8 |
| XSLTProcess | -1.833333333 | 8 |
| Zip | -5.09998724 | 8 |
| JUnitTask | -7.201200387 | 8 |
| JUnitTest | -2.5 | 8 |
| JUnitTestRunner | -3.7 | 8 |
| DBAppender | -3.6 | 8 |
| TokenStream | -5.6833092 | 8 |
| RollingFileAppender | 0.715689733 | 4 |
| LogFilePatternReceiver | -4.66663436 | 8 |
| CustomSQLDBReceiver$CustomReceiverJob | -7.633333333 | 8 |
| Tar | -0.00030152 | 5 |
| Zip | -2.716E-05 | 5 |
| JUnitTask | -8.868E-05 | 5 |
| BuildEvent | -0.033333333 | 6 |

| | | |
|---|---|---|
| EjbJar$Config | -38.23333333 | 11 |
| HistoryItem | -2.066666667 | 11 |
| Actions | -28.06666667 | 11 |
| ToDoTreeRenderer | -1.366666667 | 11 |
| ActionAggregation | -7.2559788 | 11 |
| ActionCompartmentDisplay | -7.266666667 | 11 |
| ActionMultiplicity | -9.588354667 | 11 |
| TextTool | -1.365962 | 7 |
| AbstractTool | 3.1575372 | 3 |
| CMBuilder | 4.266666667 | 2 |
| XSParticleDecl | -4.533333333 | 10 |
| XML11NSDTDValidator | -1.590049467 | 8 |
| XMLNSDTDValidator | -1.4651408 | 8 |
| XMLSchemaFactory | -2.0330456 | 8 |
| XSDFACM | -1.4971632 | 8 |
| NodeImpl | -13.26666667 | 11 |
| ElementState | -35.9 | 11 |
| DOMLocatorImpl | -9.466666667 | 11 |
| DOMErrorImpl | -11.06666667 | 11 |
| XMLSimpleType | -21.33333333 | 11 |
| XMLEntityDecl | -14.86666667 | 11 |
| HTMLSerializer | -0.000128 | 11 |

# APPENDIX Ⅱ

## Expert Evaluation Form for the Bad Smells Prioritization Model

## Bad Smells Prioritization Model (BSPM)

The aim of BSPM is to assist software managers in ordering bad smells based on their impact on maintainability. We think knowing the impact of bad smells on the maintainability is an important step. It helps in understanding the importance of bad smells and their impact on software quality. However, we believe that knowing the impact is not enough. We think that having an ordered list of bad smells based on their impact quality is of great value to practitioners.

Based on BSPM, the following steps are to be done:

1. **Elicit Bad Smells Candidate List:** In this step, we select the bad smells we want to investigate and prioritize.

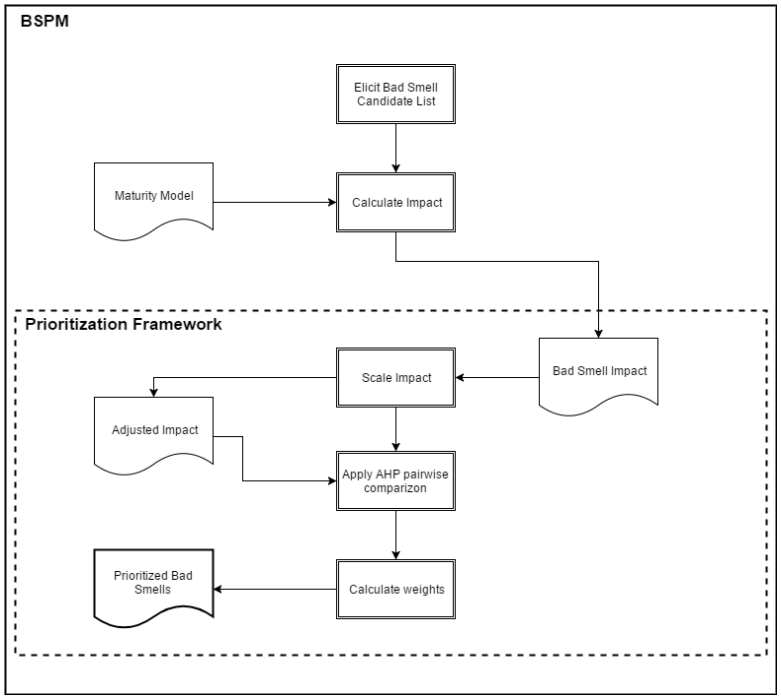
2. **Calculate Impact:** In this step, the impact of bad smell is calculated. This is done by utilizing a maturity model to measure the maintainability change before and after removing the bad smell from the code (i.e. refactoring). The result of this stage is list of bad smells and refactoring combinations *and* their average impact on maintainability.

3. **Scale Impact:** In this step, we take the bad smell impact, which is the output of the previous step, and scale it to a positive scale. The output of this step is the list of bad smells with their adjusted impact.

4. **Apply AHP pairwise comparison:** In this step, we generate a matrix to represent the relative impact of one bad smell to another. In this matrix, for each cell $a_{ij}$,its value will be

the impact of bad smell $BS_i$ divided by the impact of bad smell $BS_j$. We do this for each pair of bad smells under consideration.

5. **Calculate weights:** The final step is calculating the weight for each bad smell. This is done by normalizing any column of the matrix. The result of this step is a list of bad smells and their relative impact on maintainability.




Please take a look at the proposed model above, then rate how much you agree or disagree with the following statements:

| | Strongly Disagree | Slightly Disagree | Neutral | Slightly Agree | Strongly Agree |
|---|---|---|---|---|---|
| The BSPM model is simple | | | | | |
| The BSPM model is easy to understand | | | | | |
| The BSPM model is easy to use | | | | | |
| The BSPM model is flexible | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| The BSPM model is practical for use in industry | | | | | |
| The BSPM model is a useful tool for practitioners | | | | | |

What can be done to make the model more useful?

Do you have any suggestion to improve the model?

Your name (optional):

# References


[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code," *Xtemp01,* pp. 1-337, 1999.

[2] 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, pp. 179-202, 2011.

[3] P. Bhatt, G. Shroff, and A. K. Misra, "Influencing factors in outsourced software maintenance," *ACM SIGSOFT Software Engineering Notes,* vol. 31, pp. 1-6, 2006.

[4] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering,* vol. 30, pp. 126-139, 2004.

[5] M. Mantyla, "Bad smells in software-a taxonomy and an empirical study," *Helsinki University of Technology*, 2003.

[6] 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*, 2005, pp. 15-15.

[7] F. Van Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, 2004, pp. 336-339.

[8] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*: Springer Science & Business Media, 2007.

[9] M. V. Mantyla, J. Vanhanen, and C. Lassenius, "Bad smells-humans as code critics," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004, pp. 399-408.

[10] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, pp. 395-431, 2006.

[11] G. A. García-Mireles, M. A. n. Moraga, and F. García, "Development of maturity models: a systematic literature review," 2012.

[12] C. G. von Wangenheim, J. C. R. Hauck, C. F. Salviano, and A. von Wangenheim, "Systematic literature review of software process capability/maturity models," in *Proceedings of International Conference on Software Process Improvement and Capabity Determination (SPICE), Pisa, Italy*, 2010.

[13] J. Karlsson, C. Wohlin, and B. Regnell, "An evaluation of methods for prioritizing software requirements," *Information and Software Technology*, vol. 39, pp. 939-947, 1998.

[14] J. ali Khan, I. Q. Izaz-ur-Rehman, S. P. Khan, and Y. H. Khan, "An Evaluation of Requirement Prioritization Techniques with ANP," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 7, 2016.

[15] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002.

[16] C. J. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful: Patterns of cloning in software," *Empirical Software Engineering*, vol. 13, pp. 645-692, 2008.

[17] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings," in *Fundamental Approaches to Software Engineering*. vol. 3922, L. Baresi and R. Heckel, Eds., ed: Springer Berlin Heidelberg, 2006, pp. 411-425.

[18] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment," in *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, 2007, pp. 18-18.

[19] K. Miryung, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, 2004, pp. 83-92.

[20] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating Code Clone Genealogies at Release Level: An Empirical Study," in *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, 2010, pp. 87-96.

[21] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *SIGSOFT Softw. Eng. Notes,* vol. 30, pp. 187-196, 2005.

[22] 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, pp. 1120-1128, 2007.

[23] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?," *Empirical Software Engineering,* vol. 17, pp. 503-530, 2012.

[24] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the Impact of Design Flaws on Software Defects," in *2010 10th International Conference on Quality Software*, ed: IEEE, 2010, pp. 23-31.

[25] R. Marinescu and C. Marinescu, "Are the clients of flawed classes (also) defect prone?," *Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011,* pp. 65-74, 2011.

[26] F. Khomh, M. Di Penta, and Y. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, ed: IEEE, 2009, pp. 75-84.

[27] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1-10.

[28] 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," presented at the Proceedings of the

2009 3rd International Symposium on Empirical Software Engineering and Measurement, 2009.

[29] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software,* vol. 72, pp. 129-143, 2004.

[30] 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, pp. 2223-2242, 2013.

[31] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software,* vol. 86, pp. 2639-2653, 2013.

[32] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *Software Engineering, IEEE Transactions on,* vol. 28, pp. 654-670, 2002.

[33] T. Kamiya, "The official CCFinderX website," *URL http://www.ccfinder.net/ccfinderx.html Last accessed November,* 2017.

[34] R. Gronback, "Software remodeling: improving design and implementation quality, using audits, metrics and refactoring in Borland Together ControlCenter," *A Borland White Paper,* vol. 39, 2003.

[35] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Proceedings of 20th IEEE International Conference Software Maintenance*, 2004, pp. 350-359.

[36] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*: Springer-Verlag New York, Inc., 2005.

[37] B. Together, "Borland together," *URL http://www.borland.com/together/. Last Accessed November 2017,* vol. 197, 2017.

[38] M. A. Iqbal, A. M. Zaidi, and S. Murtaza, "A New Requirement Prioritization Model for Market Driven Products Using Analytical Hierarchical Process," in *Data Storage and Data Engineering (DSDE), 2010 International Conference on*, 2010, pp. 142-149.

[39] T. Saaty, "The analytic hierarchy process: planning, priority setting, resource allocation. 1980," *McGrawHill, New York*.

[40] A. Perini, A. Susi, and P. Avesani, "A Machine Learning Approach to Software Requirements Prioritization," *Software Engineering, IEEE Transactions on*, vol. 39, pp. 445-461, 2013.

[41] P. Avesani, S. Ferrari, and A. Susi, "Case-based ranking for decision support systems," in *Case-Based Reasoning Research and Development*, ed: Springer, 2003, pp. 35-49.

[42] P. Avesani, A. Susi, and D. Zanoni, "Collaborative case-based preference elicitation," in *Innovations in Applied Artificial Intelligence*, ed: Springer, 2005, pp. 752-761.

[43] N. Saleh, A. A. Sharawi, M. Abd Elwahed, A. Petti, D. Puppato, and G. Balestra, "Preventive Maintenance Prioritization Index of Medical Equipment Using Quality Function Deployment," *Biomedical and Health Informatics, IEEE Journal of*, vol. 19, pp. 1029-1035, 2015.

[44] D. J. Delgado- Hernandez, K. E. Bampton, and E. Aspinwall, "Quality function deployment in construction," *Construction Management and Economics*, vol. 25, pp. 597-609, 2007.

[45] R. Kavitha, V. R. Kavitha, and N. S. Kumar, "Requirement based test case prioritization," in *Communication Control and Computing Technologies (ICCCCT), 2010 IEEE International Conference on*, 2010, pp. 826-829.

[46] E. L. G. Alves, P. D. L. Machado, T. Massoni, and S. T. C. Santos, "A refactoring-based approach for test case selection and prioritization," in *Automation of Software Test (AST), 2013 8th International Workshop on*, 2013, pp. 93-99.

[47] J. P. Correia and J. Visser, "Certification of technical quality of software products," in *Proc. of the Int'l Workshop on Foundations and Techniques for Open Source Software Certification*, 2008, pp. 35-51.

[48] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, pp. 287-307, 2012.

[49] O. I. d. Normalización, *ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*: ISO, 2011.

[50] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, pp. 30-39.

[51] A. Ant, "The Apache Software Foundation [http://ant.apache.org/]," *Retrieved January 2017*.

[52] J. Robins, D. Redmiles, and D. Hilbert, "ArgoUML [http://argouml.tigris.org/]," ed: Retrieved January 2017.

[53] A. Log4j, "Log4J, The Apache Software Foundation [https://logging.apache.org/log4j/]," ed: Retrieved January 2017.

[54] E. Gamma and T. Eggenschwiler, "JHotDraw [www.jhotdraw.org/]," ed: Retreived January 2017.

[55] X. J. Parser, "Apache XML Project [http://xerces.apache.org/]," ed: Retreived January 2017.

[56] T. Saaty, "The Analytic Hierarchy Process, McGraw-Hill, New York, 1980," *There is no corresponding record for this reference*.

[57] P. T. Harker, "The art and science of decision making: The analytic hierarchy process," in *The Analytic Hierarchy Process*, ed: Springer, 1989, pp. 3-36.

[58] P. T. Harker and L. G. Vargas, "The theory of ratio scale estimation: Saaty's analytic hierarchy process," *Management science*, vol. 33, pp. 1383-1403, 1987.

[59] Y. Y. L. Helgesson, M. Höst, and K. Weyns, "A review of methods for evaluation of maturity models for process improvement," *Journal of Software: Evolution and Process,* vol. 24, pp. 436-454, 2012.

[60] J. P. Correia, Y. Kanellopoulos, and J. Visser, "A survey-based study of the mapping of system properties to ISO/IEC 9126 maintainability characteristics," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, 2009, pp. 61-70.

[61] B. Luijten and J. Visser, "Faster defect resolution with higher technical quality of software," *Technical Report Series TUD-SERG-2010-006,* 2010.

[62] O. S. Vaidya and S. Kumar, "Analytic hierarchy process: An overview of applications," *European Journal of Operational Research,* vol. 169, pp. 1-29, 2006/02/16/ 2006.

[63] J. Pérez, "Some comments on Saaty's AHP," *Management Science,* vol. 41, pp. 1091-1095, 1995.

# Vitae


| | |
|---|---|
| Name | : Turki Rujaian Alshammari |
| Nationality | : Saudi |
| Date of Birth | : 26 May 1989 |
| Email | : turki@uohb.edu.sa |
| Address | : Al Jamiah, Hafar Al Batin 39524 |
| Academic Background | : **Master of Science in Software Engineering**<br>Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Saudi Arabia<br>(2013-2017)<br><br>**Bachelor of Science in Software Engineering**<br>Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Saudi Arabia<br>(2007-2011) |