論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Supporting Reactive and Proactive Source Code Refactoring: A Context-Based Approach
著者(和文)	Sae-LimNatthawute
Author(English)	Natthawute Sae-Lim
出典(和文)	学位:博士(学術), 学位授与機関:東京工業大学, 報告番号:甲第11328号, 授与年月日:2019年9月20日, 学位の種別:課程博士, 審査員:佐伯 元司,権藤 克彦,渡部 卓雄,小林 隆志,林 晋平
Citation(English)	Degree:Doctor (Academic), Conferring organization: Tokyo Institute of Technology, Report number:甲第11328号, Conferred date:2019/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

DOCTORAL DISSERTATION

Supporting Reactive and Proactive Source Code Refactoring: A Context-Based Approach

---

Natthawute Sae-Lim

A dissertation submitted for the degree of
Doctor of Philosophy

in the

Department of Computer Science
School of Computing
Tokyo Institute of Technology

August, 2019

Summary

Software quality is a major concern in software development due to the rapidly growing demand of software system. Code smell is often used as an indicator of a design flaw or problem in source code. Code fragments containing code smell (i.e., smelly code) are handled by applying refactoring technique to improve the quality. Refactoring is a technique to restructure source code without changing its external behavior to improve source code quality. It can be divided to reactive and proactive refactoring based on timing and purpose of the operation. Reactive refactoring is the refactoring process that is applied after the source code become smelly to remove the code smells whereas proactive refactoring is the refactoring process that is applied before the source code become smelly to prevent code smell.

In issue-driven software development projects, development teams tend to adopt an issue tracking system to manage their tasks such as bug fixing or feature implementation. In this situation, we can regard modules that developers are going to modify to complete the issues as their context. Many studies have shown that developers are more likely to refactor source code that are related to their context, e.g., to prepare source code for implementation. However, most of the tools that recommend location for refactoring do not consider such a context. Therefore, developers have to perform the time-consuming process of identifying relevant outputs.

ln this dissertation, we propose an approach to support reactive and proactive refactoring recommendation by considering developers' context. This dissertation can be divided into three parts: (1) a study on factors that developers use to filter and prioritize code smells; (2) an approach to support reactive refactoring; and (3) an approach to support proactive refactoring.

(1) A study on factors that developers use to filter and prioritize code smells.

In order to improve the results of existing code smell detection tools, we conduct a study on how professional developers filter and prioritize code smells. A study is conducted with ten professional developers under the

condition that they complete a list of tasks. The results show that developers are more likely to refactor code smells that are related to their context. Therefore, the result of this part lays a foundation of this dissertation in a sense that developers' context should be considered when recommending modules for refactoring. The results from this study may also facilitate further research into code smell detection, prioritization, and filtration to better focus on the actual needs of developers.

(2) An approach to support reactive refactoring.

In this dissertation, we propose a technique to prioritize code smells using developers' context to improve the result of existing code smell detection tools. We use impact analysis techniques to predict modules that developers are likely to modify which can be regard as their context. Then, we prioritize code smells based on such information. In addition, we also study the factors that affect the performance and the optimized parameters of the technique. The proposed technique is evaluated using open source software projects and a controlled experiment with professional developers. The results show that our technique can provide more relevant results that are aligned with how developers actually prioritize their code smells.

(3) An approach to support proactive refactoring.

In order to identify the target for proactive refactoring, we propose a concept of decaying modules which are non-smelly modules but are getting closer to become code smells. Proactively refactoring such modules can prevent source code from becoming smelly. In this dissertation, we present an investigation on the characteristics of decaying modules. Furthermore, we propose an approach to use a machine learning technique to predict decaying modules which can be used to support developers during refactoring strategy planning. The prediction approach uses source code quality metrics as predictor variables and whether a module will get decayed in the next release as a response variable. We also study the use of developers' context to improve the performance of the prediction model. The developers' context can be estimated by applying impact analysis technique to change descriptions and source code.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor, Professor Motoshi Saeki, for the continuous support of my Ph.D. study. I cannot imagine my student life without his valuable advice and encouragement.

I am very grateful to Professor Shinpei Hayashi not only for teaching me how to do research, but also inspiring, challenging, and guiding me throughout the years.

Further, I would like to thank the co-author of my publications, Aoi Takahashi, for his contribution to my research life. My sincere thanks also go to all members of Saeki and Hayashi laboratory for such amazing working environment and all the fun we have had in the last five years.

This work would not be possible without the financial support from Japanese Government (Monbukagakusho: MEXT) Scholarship.

Last but not least, I would like to express my heart-felt gratitude to my friends and family for their support and encouragement whenever I needed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software quality becomes a primary concern in software development since the demand for system software is rapidly grown. A technique that is often used to improve the quality of source code is a refactoring technique. Refactoring refers to the technique of restructuring the source code while maintaining the external behavior of the system [1]. In practice, the locations of source code that are advised to apply refactoring technique are the modules containing code smells. Code smell is often used as an indicator of the design flaw or problem in the source code, which can affect software quality such as understandability and maintainability [1, 2, 3]. For instance, God Class is a code smell indicating a class that is overly large and complex, which tends to have too many functionalities [3]. For example, we can apply extract class operation to a class with God Class code smell to create a new class and move related attributes and methods to it.

In modern software development project, development teams tend to adopt an issue tracking system such as Jira[1] or Bugzilla[2] to manage their lists of issues. The issue can be anything related to software change requests, e.g., bug fixing or feature implementation. A number of studies have found that the refactoring activity of developers is mostly driven by such change requirements, rather than the existence of code smells themselves [4, 5, 6, 7]. In addition, Meng et al. [8] found that developers do not typically refactor their code unless they must change the code to fix some bugs or introduce new features. Bavota et al. [9] supported this statement by asserting the possibility that the only goal of refactoring, for developers, is to prepare the source

[1] `https://www.atlassian.com/software/jira`
[2] `https://www.bugzilla.org/`

TABLE 1.1: Comparison between reactive and proactive refactoring

	Reactive Refactoring	**Proactive Refactoring**
Timing:	**After** code becomes smelly	**Before** code becomes smelly
Purpose:	To **remove** code smells	To **prevent** code smells
Metaphor:	Tooth decay treatment	Brushing teeth

code for future changes. Given that situation, modules within the focus of the developers can be regarded as their context. Such modules are likely to have higher priorities to fix code smells and improve code quality than others because fixing code smells in them might contribute to the support of future implementation, i.e., improving the understandability and extendibility of the source code. In this dissertation, we use the definition of *context* similarly to *task context* defined by Kersten and Murphy as "the information–a graph of elements and relationships of program artifacts–that a programmer needs to know to complete that task" [10].

However, although many techniques have been proposed to detect code smells and recommending refactoring opportunities [11, 12, 13], most of them ignore such a context and output too many results. Also, most of the tools output the results only from analyzing the whole source code without prioritizing the results or prioritizing them with factors that are not related to the context of the developer, e.g., the severity of each smell, which represents how strong a code smell is. Therefore, the time-consuming process of identifying relevant outputs is left to developers, which is similar to the fact that static analysis tools are not used well by developers because of numerous detected warnings [14].

Besides, in this study, we divide refactoring into two categories: reactive and proactive refactoring depends on the timing and purpose of the operation. Table 1.1 summarizes the differences between reactive and proactive refactoring. Reactive refactoring refers to the refactoring process that is applied after the source code becomes smelly with the purpose of removing code smells whereas proactive refactoring refers to the refactoring process that is applied before the source code becomes smelly with the purpose of preventing code smells. Reactive refactoring is comparable as tooth decay treatment where the patients take care of their tooth decays after they happen. On the other hand, proactive refactoring can be compared to the act of

brushing one's teeth every day to remove small food particles and preventing teeth from getting decayed.

Nevertheless, most of the existing refactoring recommendation techniques are designed to support only reactive refactoring. A distinct advantage that such tools allow developers to focus on the most problematic part of the source code (smelly code) rather than handling every part of the source code. However, an essential shortcoming of such approaches is that developers cannot prevent code smells from occurring. In other words, when the tools warn developers of the code smells, they have already suffered the harmful consequences of the code smells (e.g., low understandability and maintainability). Therefore, proactive refactoring should also apply to the source code in order to maintain good source code quality.

1.2 Approach

To solve these problems, in this dissertation, we propose a technique to utilize developers' context to support source code refactoring. The technique can be used to support both reactive and proactive refactoring. For reactive refactoring, we propose a technique to prioritize code smells from code smell detectors by considering developers' current context. As described earlier, this technique specifically examines support of the issue-driven software development projects where development teams tend to adopt an issue tracking system to manage their lists of issues. Using the proposed technique, change descriptions in an issue tracking system that are going to be implemented using a particular milestone are used to estimate the developers' context. Such changes are going to be implemented by modifying or extending existing modules in the source code. These existing modules can be located using impact analysis [15, 16]. We regard such relevant modules as their context because they are likely to be the location for implementation of the new features in the change descriptions. As a result, code smells that appear in relevant modules should be assigned higher priority so that the developers can readily distinguish them from irrelevant code smells. With support from our technique, developers can obtain a prioritized list of code smells based on their relevance to the context.

For proactive refactoring, first, we propose a concept of decaying modules which are non-smelly modules but are getting closer to become smelly.

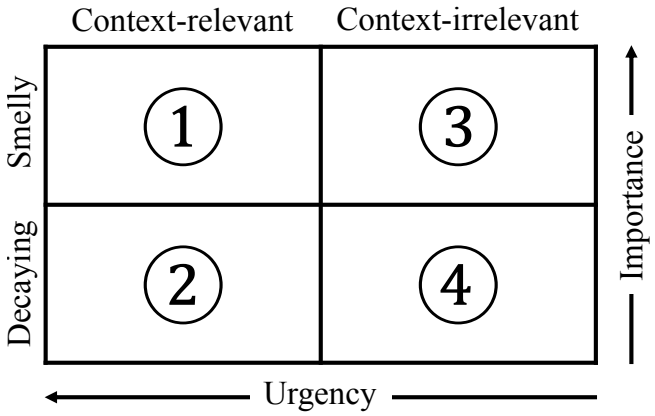


FIGURE 1.1: Quadrant analysis of refactoring target prioritization.

The decaying modules can be detected using historical information with the similar criteria used to detect code smells. The decaying modules can be used to warn developers of the modules that are about to become smelly so that they can take preventive measures rather than waiting until the source code becomes smelly. Then, to facilitate developers' refactoring strategy, we propose a machine learning technique to predict decaying modules. To improve the performance of the prediction model, we also propose the use of developers' context to enhance the performance of the prediction model under the assumption that modules that developers are going to modify are more likely to get decayed than the modules without modification. Similarly to reactive refactoring, the context of developers is estimated by applying impact analysis techniques to change descriptions. Consequently, the result of the prediction model can be used to support developers when planning for proactive refactoring.

1.3 Goal

The final goal of our approach is the system that recommends modules for refactoring based on the guideline presented using quadrant analysis in Fig. 1.1. The vertical axis represents the importance of the modules, i.e., smelly modules are more important than decaying ones. This is because smelly modules have bad effect on the system while decaying modules are yet to have such an effect; therefore, they can be considered to be less important. The

horizontal axis represents the urgency of the modules, i.e., context-relevant modules are more urgent than context-irrelevant ones. The main reason is that although solving code smells that are irrelevant to developers' context (e.g., modules developers plan to modify) may improve the overall quality of the system, it does not support the developers' current activities. This statement is also supported by previous studies on professional developers showing that developers tend to refactor the code smells related to their context. Therefore, context-irrelevant modules can be postponed until they become relevant to the developers' context. Considering the importance and urgency together, it is evident that smelly context-relevant modules should have the highest priority, and decaying context-irrelevant modules should have the lowest priority. However, between decaying context-relevant and smelly context-irrelevant modules, we argue that decaying context-relevant modules should be given higher priority than smelly context-irrelevant modules. As aforementioned, solving context-irrelevant modules, irrespective of their quality, is not likely to facilitate planned implementation. On the contrary, in addition to preventing modules from being affected by code smells, solving decaying context-relevant modules can help developers get ready for their implementation.

1.4 Problem Statements

Even though the research community has put many efforts into proposing tools for detecting code smells, many aspects can still be improved. The aspects that we focus on in this study can be summarized as follows.

1. **A lack of empirical data on how developers handle code smells hinders the improvement of code smell detection approaches.**

   Despite a number of code smell detection approaches, there are still needs for better approaches from practitioners [17]. This may suggest that existing approaches are not aligned with practitioners' actual needs. Nevertheless, developing these approaches to align with actual needs of practitioners is difficult due to a lack of empirical evidence regarding how practitioners handle code smells in practice. Hence, in this dissertation, we study the factors that professional developers use to handle, i.e., filter and prioritize, code smells. This result could help

the research community improves the practicability and usability of the approaches regarding code smells in the future.

2. **Most of the existing code smell detection approaches do not consider the developers' context.**

   Even though it is advisable to remove code smells from the system, many studies have found that developers only refactor their source code unless it is related to their context, i.e., they have to fix a bug or implement a new feature [7, 8, 9, 18]. Nevertheless, most of the existing approaches do not consider such a context when detecting code smells. To this end, we propose an approach to prioritize code smells based on developers' context. We expect our approach to support developers when considering code smells that should be removed.

3. **Existing code smell detectors do not allow developers to prevent code smells from occurring.**

   One significant drawback of existing code smell detectors is that they consider a code smell detection as a binary classification. In other words, modules in source code are classified in either a clean module or a smelly module. As a consequence, developers cannot track the progression of code smells or prevent them from occurring. Thus, in this study, we propose a concept of a decaying module which represents a non-smelly module that its quality is getting worsened and is likely to become smelly. This approach can help developers track the progression of code smells and prevent them from occurring in the system.

4. **There is no technique to support proactive refactoring.**

   As discusses earlier, proactive refactoring refers to a process of refactoring source code to prevent code smells. However, a lack of technique that supports proactive refactoring prevents practitioners from performing proactive refactoring because most of the refactoring recommendation techniques focus on reactive refactoring. To support proactive refactoring process, we propose a machine learning based technique to predict decaying modules. This technique can be used to support developers when planning for a refactoring strategy.

1.5 Contributions

The main contribution of this research is that we propose an approach to use developers' context to support source code refactoring. Specifically, the major contributions of this research can be summarized as follows.

1. **A report on the factors that developers use to filter and prioritize code smells.**

   In order to improve the results of existing code smell detection tools, we conduct a study on how professional developers filter and prioritize code smells. The result shows a list of factor that developers use as criteria when preparing for prefactoring. Most importantly, it shows evidence that developers tend to refactor source code that is related to their context, which is supported by many works in the literature. Therefore, the result of this study is a foundation of this dissertation, where we propose techniques that are based on utilizing developers' context. The details of this study can be found in Chapter 3.

2. **A context-based code smells prioritization approach.**

   As described earlier, we propose an approach to prioritize code smells using developers' context. The context is estimated by applying impact analysis to change descriptions in software development projects. We evaluate our approach by both using open source software projects and conducting a controlled experiment with professional developers. Furthermore, we also present investigations related to the proposed technique, such as the factors that affect the performance and the optimized parameters of the technique. The details of this approach can be found in Chapters 4 and 5.

3. **A concept of decaying modules to support proactive refactoring.**

   We introduce the concept of decaying modules, which represents modules that are getting closer to become code smells. The concept of decaying modules can be used to support proactive refactoring, which refers to the process of refactoring non-smelly modules to prevent them from becoming smelly. Additionally, we also present an investigation of the characteristics of decaying modules. The details of decaying modules are described in Chapter 6.

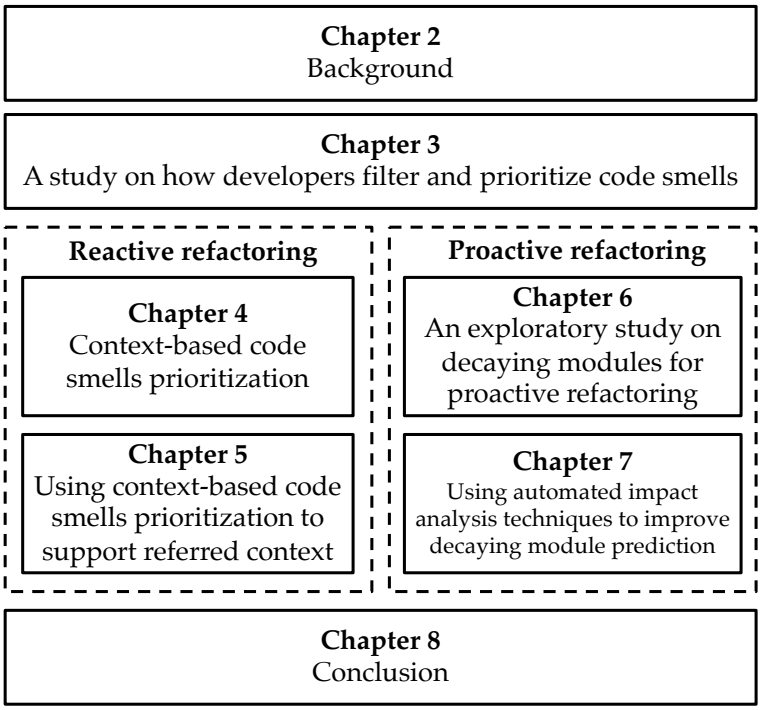


FIGURE 1.2: Dissertation structure.

4. **A machine learning based approach to predict decaying modules.**

   To support developers' refactoring planning, we propose an approach to use a machine learning approach to predict decaying modules. The approach mainly uses source code quality metrics as predictor variables to predict a response variable, whether a specified module will get decayed in the next release. Additionally, we also propose an approach to improve prediction model performance using developers' context. The context is estimated by applying an impact analysis technique to change descriptions similarly to code smells prioritization approach. An evaluation on open source projects shows the effectiveness of our approach. The details of this approach is described in Chapters 6 and 7.

1.6 Dissertation Structure

Figure 1.2 shows the structure of this dissertation. Each chapter can be briefly summarized as follows.

Chapter 2 briefly introduces the background knowledge in this dissertation. We first present the background of code smell and refactoring. We summarize the software change process, especially, the prefactoring phase, which is the main phase that this dissertation aims to support. In addition, we describe impact analysis approaches, including approaches that use information retrieval technique, dynamic analysis technique, and approaches that combine different techniques together.

Chapter 3 describes the study on how developers filter and prioritize code smells. This chapter lays the foundation of this dissertation as we found that the most important factor used by developers when filtering and prioritizing code smells is the relevance to their context.

Chapter 4 presents the context-based code smells prioritization approach. This chapter also includes empirical studies on open source projects to evaluate our approach and study factors that affect performance. A Controlled experiment is also conducted to evaluate our approach with professional developers.

Chapter 5 describes the study on using the context-based code smells prioritization approach to support referred context. The evaluation in Chapter 4 shows that our approach can be used to support developers' on the modules they intend to make changes (i.e., modified context). However, before making actual changes, developers usually browse through many modules and refer to them (i.e., referred context). Thus, it is essential to confirm that our approach can support not only modified context but also referred context.

Chapter 6 introduces the idea of decaying modules that can be used to support proactive refactoring. The formal definition of a decaying module is presented to represent a non-smelly module that is likely to become smelly. To understand the characteristics of decaying modules, we also present an empirical study on open source projects. Furthermore, this chapter also includes the study of using a machine learning technique to predict decaying modules.

Chapter 7 presents a study on using developers' context to improve decaying modules prediction. The developers' context is estimated by using a similar approach to Chapter 3. We also discuss future directions to improve the performance of the prediction model.

Chapter 8 concludes this dissertation. We briefly review our techniques in earlier chapters. Finally, we discuss further directions that can be explored.

Chapter 2

Background

2.1 Code Smell

Code smell is an indicator of design flaws or problems in the source code [1, 2, 3]. Table 2.1 shows examples of code smells with their short description. For example, the first code smell is called God Class in a class level component which tends to centralize functionality of the system. Code smells of many types are summarized as smell catalogs [1, 2, 3]. Code components that are affected by code smells are recommended to improve the quality since they are likely to cause severe problems in the future as discussed in the literature, e.g., making source code more difficult to understand and maintain, or increasing code component's fault-proneness. Developers can do so by removing the code smells in the components. Moreover, Yamashita et al. [19] found that code smells could affect important maintainability aspects. Hermans and Aivaloglou [20] conducted a controlled experiment and found that code smells influence the block-based programming performance of novice programmers.

Many researches have been done to investigate factors that could introduce code smells into the system. Tufano et al. [21] conducted a large-scale empirical study and found that the main activities which tends to introduce code smells are implementing features and enhancing existing ones. Moreover, they found that developers with high workloads and pressure are more likely to introduce code smells to the system. Vale et al. [22] investigated the factors that could influence code smells based on a software developer's perspective and found that factors such as time pressure, bad design decisions, lack of business knowledge, inexperienced developers, no time for refactoring, and low priority to software quality influence code smells.

TABLE 2.1: Examples of code smells

Type	Granularity	Short description
God Class	Class	A class that centralizes functionality of the system. A God Class violate the principles of object-oriented design that one class should focus on one responsibility, resulting in very large and complex class.
Data Class	Class	A class that contains attributes without main functionality. These attributes tend to be used by other classes which are against the encapsulation principle of object-oriented programming.
Feature Envy	Method	A method that tends to access methods of other classes. This kind of methods are likely to be misplaced and should be moved to an appropriate class.
Brain Method	Method	A method that contains most of the functionality of the class. A Brain Method tends to be long and complex.
Refused Parent Bequest	Class	A subclass that does not need all of the methods or attributes from its parent class. It is a signal of a wrong hierarchy of classes and likely to cause problems and confusion.
Duplicated Code	Method	A pair of operations that contain the same expression. It affects the uniqueness of the entities and makes it difficult to find the actual location of the implementation.

TABLE 2.2: Examples of refactoring operations

Type	Granularity	Short description
Extract Class	Class	Create a new class and move related attributes and methods that are more related to the new class in it.
Move Method	Method	Create a new method in the class that it should belong to and move the content of methods to the new method.
Replace Magic Number with Symbolic Constant	Method	Create a constant that represent the magic number and replace the number with it.
Replace Inheritance with Delegation	Class	Create a field for the superclass, delegate methods to the new object, and remove inheritance.
Parameterize Method	Method	Create a method that takes a parameter instead of having multiple methods with similar functionality with different values.

Figure 2.1 shows an example portion of Employee class which is affected by several code smells. The most noticeable code smell, in this case, is the Duplicated Code. As we can see in the figure, there is the same code fragment in both `printSalary()` method and `printIncomeTax()` method even though the purpose of this code fragment in both methods is to calculate the salary of an employee. Consequently, it reduces several factors that reflect code quality, e.g., understandability and extensibility. For understandability, this makes the methods unnecessarily long and more difficult to read. For extensibility, a developer would have to unnecessarily waste effort modifying two code fragments if there are changes about a method of calculating salary. In the worse case, the developer is not aware of another location and make changes only in one location. This case is likely to cause inconsistency in the program which consequently lead to misfunction of software or a so-called *bug*.

```
public class Employee...
   private void printSalary()
   {
      int salary;
      if (workingHours > 40)
      {
         salary = (40 * 1000) + ((workingHours - 40) * 1500);
      }
      else
      {
         salary = workingHours * 1000;
      }
      System.out.println("Salary: " + salary);
   }

   private void printIncomeTax()
   {
      int incomeTax;
      int salary;
      if (workingHours > 40)
      {
         salary = (40 * 1000) + ((workingHours - 40) * 1500);
      }
      else
      {
         salary = workingHours * 1000;
      }
      incomeTax = salary * 0.08;
      System.out.println("Tax: " + incomeTax);
   }
```

FIGURE 2.1: Employee class with code smells.

2.1.1 Refactoring

Refactoring, a technique for improving the software structure without changing its external behavior, is considered as the effective way to solve code smells [1]. Developers may use different refactoring operations to solve different types of smells depending on the characteristic of each one. For example, code smells related to the size of code elements such as God Class or Brain Method, can be solved by using the refactoring operations that reduce the size of code elements such as Extract Class or Extract Method. On the other hand, code smells that are related to the hierarchy of classes such as Refused Parent Bequest can be handled by refactoring operations that reorganize the relationship between subclass and super class such as Replace Inheritance with Delegation. Table 2.2 shows examples of refactoring operations and their short descriptions. The full catalog of refactoring operations can be found in the book by Fowler [1].

As we discussed previously, Employee class in Fig. 2.1 is affected by Duplicated Code code smell and should be refactored to remove it. The most appropriate refactoring operation, in this case, is Extract Method, which turns code fragment into a method. Figure 2.2 presents Employee class after applying Extract Method refactoring operation. As we can see, the duplicate code fragments are now extracted to the `getSalary()` method. Moreover, `printSalary()` and `printIncomeTax()` methods are now calculating salary by calling the `getSalary()` method instead of calculating on their own. The two methods are now shorter and easier to read, i.e., the understandability is increased. If a developer has to modify the calculating salary logic, he or she can simply do it in `getSalary()` method since the code is now centralized in there. That is to say, the extensibility of the source code is now improved.

We can further refactor the Employee class to improve the structure of this class. For example, first we apply the Extract Method operation to extract the code fragment that calculates income tax so that other methods can use when needed and the `printIncomeTax()` method can focus on one functionality. Then, we remove the unnecessary temporary variable. Finally, we apply Replace Magic Number with Symbolic Constant operation by extracting all of the magic numbers to the constant variable that represents their value. Figure 2.3 represents the Employee class after applying refactoring operations. We can see that, while keeping the same external behavior of the system, the quality of the source code is now improved, i.e., the methods

```
public class Employee...
   private void printSalary()
   {
      int salary;
      salary = getSalary();
      System.out.println("Salary: " + salary);
   }

   private void printIncomeTax()
   {
      int incomeTax;
      int salary;
      salary = getSalary();
      incomeTax = salary * 0.08;
      System.out.println("Tax: " + incomeTax);
   }

   private int getSalary()
   {
      int salary;
      if (workingHours > 40)
      {
         salary = (40 * 1000) + ((workingHours - 40) * 1500);
      }
      else
      {
         salary = workingHours * 1000;
      }
      return salary;
   }
```

FIGURE 2.2: Employee class after applying Extract Method refactoring.

```
public class Employee...
   static final int MAXIMUM_WORKING_HOURS = 40;
   static final int NORMAL_RATE = 1000;
   static final int OVERTIME_RATE = 1500;
   static final double INCOME_TAX_RATE = 0.08;
   private void printSalary()
   {
      System.out.println("Salary: " + getSalary());
   }

   private void printIncomeTax()
   {
      System.out.println("Tax: " + getIncomeTax());
   }

   private int getSalary()
   {
      if (workingHours > MAXIMUM_WORKING_HOURS)
      {
         return (MAXIMUM_WORKING_HOURS * NORMAL_RATE) +
                ((workingHours - MAXIMUM_WORKING_HOURS) * OVERTIME_RATE);
      }
      else
      {
         return workingHours * NORMAL_RATE;
      }
   }

   private double getIncomeTax()
   {
      return getSalary() * INCOME_TAX_RATE;
   }
```

FIGURE 2.3: Employee class after applying several refactoring operations.

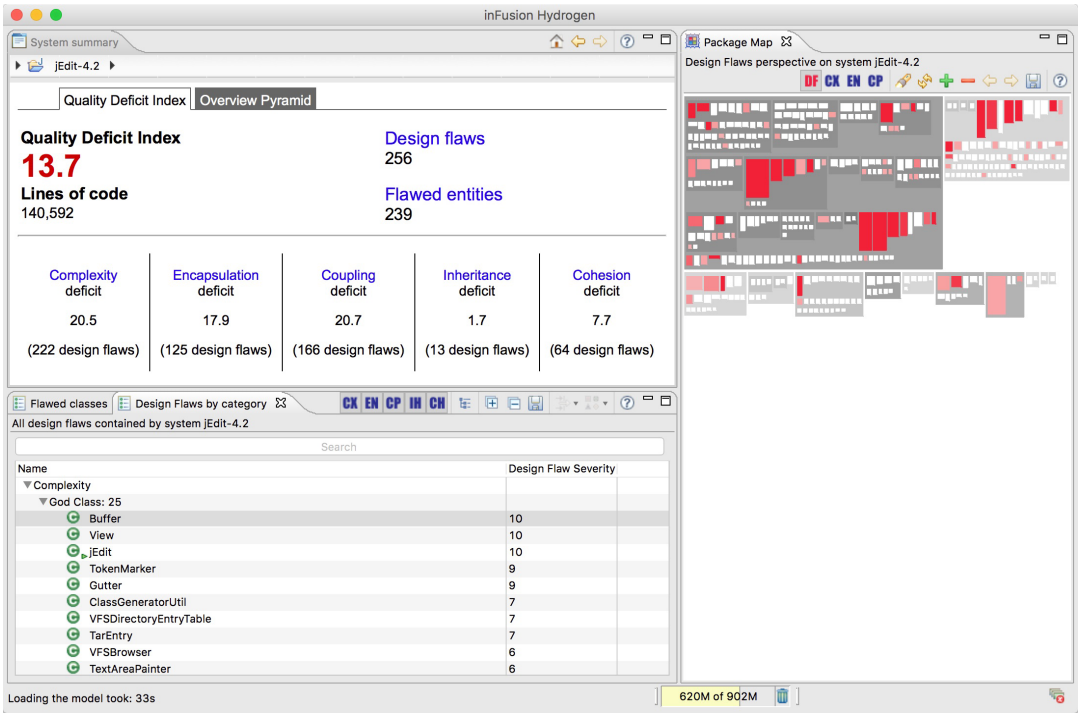


FIGURE 2.4: Code smell detection result from inFusion.

are now easier to comprehend and implement comparing to the one before refactoring.

2.1.2 Code Smell Detection

Since code smell was defined using descriptive language, the task of detecting code smells remained challenges for software developers. Therefore, many code smell detection strategies and tools have been proposed to support the task of detection automatically [11, 12]. For example, Moha et al. [23] proposed a technique to specify and detect code smells called DECOR. Palomba et al. [24] proposed a technique to detect code smells using change history information from a version control system. They also proposed a textual-based technique for detecting code smell by applying information retrieval technique to measure the probability of a code component affected by a given code smell [25]. Furthermore, techniques using machine learning were also proposed to detect code smells [26, 27, 28, 29]. Another technique for detecting code smells is to used the object-oriented code metrics. Lanza

and Marinescu [3] presented a method of using logical condition of code metrics to detect code smells. For example, in order to detect the God Class, three metrics: ATFD (Access to Foreign Data), WMC (Weight Method Count), and TCC (Tight Class Cohesion) are used to determine whether a specific class is affected by God Class code smell. In this case, a God Class code smell is determined by the rule:

$$\text{God Class} = (ATFD > \text{FEW}) \wedge (WMC \geq \text{VERY HIGH}) \wedge (TCC < 1/3). \quad (2.1)$$

Figure 2.4 shows an example of code smell detection result from inFusion [30] of jEdit[1] ver. 4.2. The top left screen shows the summary of the system, e.g., the overall quality score, the number of code smells, the number of classes affected by code smells, and code smells in each category. The right-hand side of the screen shows the package map of the system. It visualizes the location of code smells in the system. The red or white color box represents each class of the system. The deeper red color, the more code smells are in that class. The white color indicates that there is no smell in the class. The outer box represents the package of the system. A list of code smells will appear when hold a mouse over each node. Finally, the top left screen shows the list of classes that are affected by each type code smells together with its severity, representing how strong a given code smell is.

2.2 Software Change

Rajlich [15] has divided the process of software change into phases as shown in Fig. 2.5. Software developers follow this process starting from when they get change requirements until the change process is finished.

The process starts with *initiation* and then moves to the *concept location* phase where developers identify the code component that needs to be modified in order to satisfy change requirements, e.g., bug fixing or feature implementation. Developers may use system knowledge to identify it or use some automated techniques such as feature location technique to assist during the process.

After finishing the concept location phase, developers move on to the *impact analysis* phase where the full set of modules that are likely to be affected

[1] `http://www.jedit.org/`

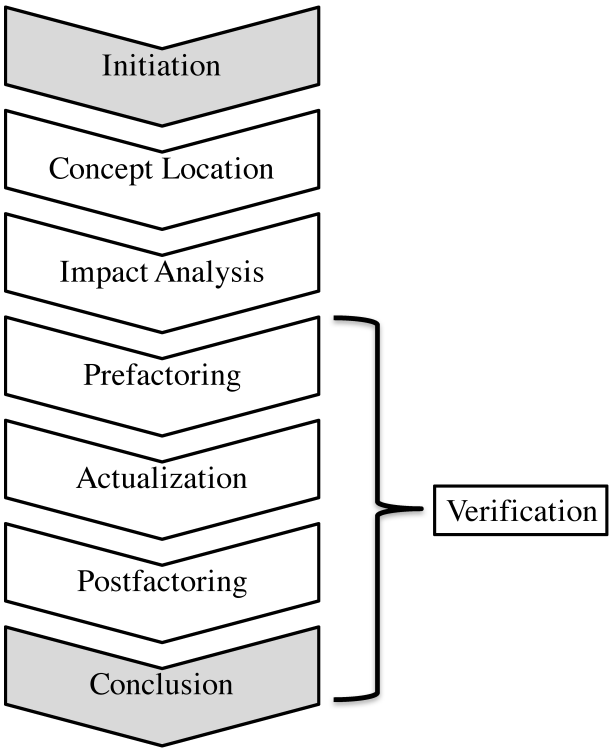


FIGURE 2.5: Software change process. Modified from [15].

by the change are identified. The reason behinds two different phases is that in the case of medium or big software system, location for making changes are not always in one location, but rather spread across modules. Developers may use the module identified by concept location phase as a starting point and then look for the related modules that are likely to have the impact. They also can use an automated technique such as information retrieval to assist impact analysis phase as well.

After deciding the modules needed to be modified, developers move on to *prefactoring* phase. In prefactoring phase, refactoring technique, a technique to improve source code structure without altering its external behavior, is applied to the target code components in order to prepare the source code for actual implementation, e.g., localize the modules that should be changed together to affect fewer modules during modification. Prefactoring can also help improving the understandability and extendibility of the source code which can facilitate developers in actualization phase.

Actualization phase is where real modification according to change requirement is implemented. Then, developers apply refactoring techniques

again in the *postfactoring* phase with the intention of cleaning up the source code that has been modified in actualization phase.

Verification phase cover prefactoring, actualization, postfactoring, and conclusion phase. The aim of this phase is to control the quality of the work done during the covered phase. Any bugs or problems that occurred may be identified and resolved during this verification phase.

Conclusion phase is the last phase in software change process where developers finish the change up, e.g., updating documentation or committing changes to version control system of the project.

2.3 Prefactoring

As explained in the earlier section, the prefactoring phase is where a developer applies the refactoring technique to the target modules in order to support their upcoming implementation. Assuming that there was a developer who wanted to implement some features in class `A`. He or she went to class `A` and found that class `A` is affected by a Blob Class code smell. That is to say, class `A` is very large and complex. Thus, implementing some features in class `A` would be difficult and take more time and effort than usual implementation since it is very difficult to understand. Then, the developer would have a choice to refactor class `A` first so that the smell Blob Class is gone. After that, the class `A` would now become very easy to understand. Therefore, implementing the feature in the cleaned class `A` will now take less time and effort.

2.4 Impact Analysis

As mentioned in the earlier section, impact analysis is the phase where the full set of modules that are going to be affected by the change are identified. According to Rajlich [15], the input of impact analysis is the modules from concept location technique, and the output is the list of related modules that are likely to be affected by the change. However, in this research, we used a change description from an issue tracking system as an input and used the list of methods obtaining from impact analysis as an output. Even though the input might be different from what defined by Rajlich [15], but the purpose of the impact analysis, which is to generate a full set of related modules, is still

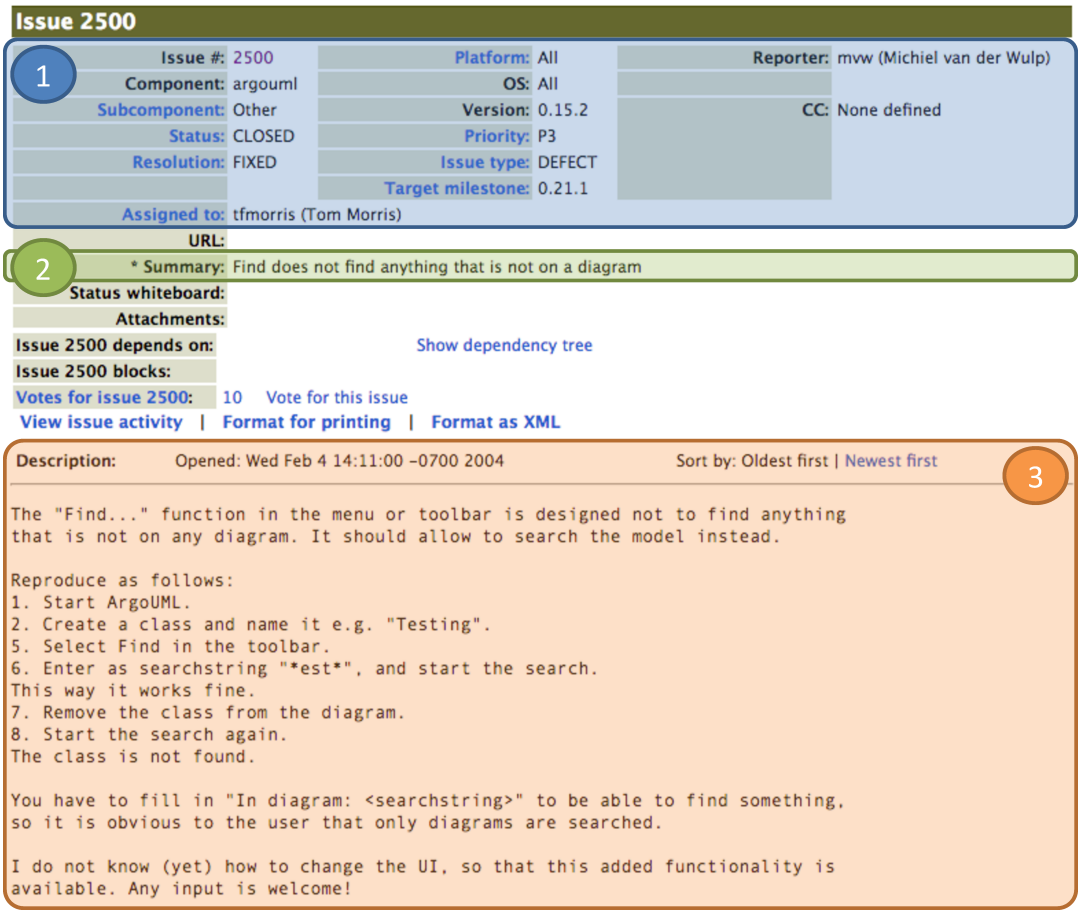


FIGURE 2.6: Example portion of an issue in issue tracking system.

the same. This section explains the automated impact analysis techniques that can be used to support developers in this phase.

2.4.1 Information Retrieval

In an issue-driven software development projects, a development team often adopt an issue tracking system such as Jira[1] or Bugzilla[2] as a tool for managing change requirements. An issue is normally composed of many detailed items, e.g., reporter, assignee, or status, but in this section, we are going to

[1] https://www.atlassian.com/software/jira
[2] https://www.bugzilla.org/

TABLE 2.3: An example portion of results from information retrieval

Method	Score
org.argouml.util.ExprSeparatorWithStrings.endChar(char)	0.69
org.argouml.util.ExprSeparatorWithStrings.reset()	0.69
org.argouml.util.ExprSeparatorWithStrings.ExprSeparatorWithStrings()	0.62
org.argouml.util.QuotedStringSeparator.QuotedStringSeparator(char,char)	0.38
org.argouml.util.QuotedStringSeparator.QuotedStringSeparator(char,char,char)	0.38
org.argouml.persistence.XmlInputStream.readAttributes()	0.34
org.argouml.ui.cmd.GenericArgoMenuBar.initMenuCreate()	0.29
org.argouml.util.QuotedStringSeparator.endChar(char)	0.28
org.argouml.uml.diagram.ProjectMemberDiagram.getDiagram()	0.28
org.argouml.uml.diagram.ProjectMemberDiagram.setDiagram(ArgoDiagram)	0.28

focus on summary or title of the issue and the description of the issue. Figure 2.6 presents an example of an issue in issue tracking system of ArgoUML project [1]. The information of the issue is shown in (1) blue area. (2) The green area shows the summary or title of the issue, and the description together with reproduction steps are shown in (3) orange area.

Issues are created either by software users, who want to report bugs or ask for feature improvement, or members of the development team who decide to change the functionality. Then, during the development process, developers would pick the issue that they are responsible for from the issue tracking system and implement changes according to the description and details specified in the issue.

As we can see, the issue in the issue tracking system plays important roles in this sort of software development process. The details in the issue contain very important clues of the context of developers, i.e., what they have to do, which modules they have to implement, and what kind of logic they have to implement or change. Therefore, it would be beneficial to use such information to assist developers during the impact analysis phase.

Information retrieval works by calculating the textual similarity between queries, e.g., change descriptions in issue, and artifacts, e.g., source code in modules. Source code, in this case, considers every part that appears as a text, e.g., variable names or comments. The higher similarity between change descriptions and source code in modules, the higher possibility of implementing the changes in that modules. Table 2.3 displays an example portion of results when apply the summary and description of the issue in Fig. 2.6 to

[1] http://argouml.tigris.org/issues/show_bug.cgi?id=2500

the information retrieval technique. The next paragraphs present overviews of the respective techniques we used for this dissertation.

Vector Space Model (VSM)

VSM is a model used mostly for information retrieval by representing documents and queries as vectors [31]. Then, the relevance of documents and queries is obtainable by calculating their mutual cosine similarity. For impact analysis, documents are source codes of a specific project; queries are change requests from the issue tracking system. The results then are the relevance between source code and change requests.

Latent Semantic Indexing (LSI)

LSI [32] is based on VSM, with the intention to handle the situation of synonymy, a group of words that share similar meanings, and polysemy, words that have multiple meanings. Actually, LSI is used for assessing the similarity between documents from the common words that two documents contain rather than simple terms. The more common words they have, the more similar they are. Details of the techniques we used for this study can be found in an earlier report of the literature [33].

Okapi BM25 (BM25)

BM25 [34] is a text-retrieval algorithm used to estimate the relevance of documents and a query. It is a derivation from the probabilistic information retrieval and often used in general purpose search engine. BM25 uses term frequency (TF), inverse document frequency (IDF), and document length (DL) to calculate the relevance of documents and a query. A study has found that BM25 performs the best when used to calculate the relevance between source code and change requests [35].

2.4.2 Dynamic Analysis

An execution trace is the information recording when running software. There are several technologies to collect the information, e.g., Java Platform Debugger Architecture (JPDA)[1] or Test and Performance Tools Platform (TPTP)[2]. The important part of execution trace for impact analysis is the list of modules that have been executed during the process. Execution trace can be obtained by software developers who are debugging the source code. Developers could also follow the reproduction step in change descriptions and record execution trace in order to obtain relevant modules. Since the list contains only the modules that were executed during the reproduction step, the modules that could be the target for implementing changes are likely to be in the list of the execution trace. Therefore, we can also use this kind of dynamic analysis to help identify relevant modules during impact analysis phase.

2.4.3 Combining Different Techniques

Gethers et al. [33] have proposed a technique to combine different impact analysis techniques in order to obtain higher accuracy. For example, after getting a list of modules from information retrieval technique, we can use the list of modules from dynamic analysis to filter out the result since the modules that are not in the result of dynamic analysis are not likely to relate to the change description. Therefore, we could remove unrelated modules and obtain higher accuracy from doing so.

[1] `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/`
[2] `https://projects.eclipse.org/projects/tptp.platform/`

Chapter 3

A Study on How Developers Filter and Prioritize Code Smells

Summary

Code smells are indicators of design flaws or problems in the source code. Various tools and techniques have been proposed for detecting code smells. These tools generally detect a large number of code smells, so approaches have also been developed for prioritizing and filtering code smells. However, lack of empirical data detailing how developers filter and prioritize code smells hinders improvements to these approaches. In this study, we investigated ten professional developers to determine the factors they use for filtering and prioritizing code smells in an open source project under the condition that they complete a list of five tasks. In total, we obtained 69 responses for code smell filtration and 50 responses for code smell prioritization from the ten professional developers. We found that *Task relevance* and *Smell severity* were most commonly considered during code smell filtration, while *Module importance* and *Task relevance* were employed most often for code smell prioritization. These results may facilitate further research into code smell detection, prioritization, and filtration to better focus on the actual needs of developers.

This chapter is based on our journal paper "An Investigative Study on How Developers Filter and Prioritize Code Smells" [36] published in IEICE Transactions on Information and Systems, Copyright © 2018 IEICE. The paper is an extended version of our earlier work "How Do Developers Select and Prioritize Code Smells? A Preliminary Study" [37] appearing in the 33rd International Conference on Software Maintenance and Evolution (ICSME'17), Copyright © 2017 IEEE.

3.1 Introduction

Code smells were first defined to represent problems in source code possibly caused by bad design decisions [1, 2]. This definition mostly applies to descriptive languages, so many studies have aimed to interpret these smells in a formal manner. In particular, several previous techniques use source code metrics to detect code smells [3, 23, 38, 39, 40]. In addition, many attempts have been made to detect code smells using other information such as historical data [24, 25].

However, results from such techniques are numerous because of the large volume of source code. Therefore, the task of screening code smells is left to developers as they are unlikely to be able to solve all code smells due to time constraints. The task of screening code smells can be divided into two steps. The first step deals with code smell filtration, i.e., developers select only a subset of code smells that they think should be addressed at the moment. Subsequently, they prioritize the results of the filtration process in the second step, i.e., decide the order in which the code smells should be handled. To support this task, many code smell filtration and prioritization techniques have been proposed. The techniques have focused on different methods such as severity-based techniques where code smells with higher severity are given higher priority [41, 42, 43], context-based techniques where code smells are prioritized according to the specific contexts of developers [44], or by using combinations of multiple factors to prioritize code smells [45, 46, 47].

Even though the research community has developed several approaches to support the code smell detection process, there is still a need for better tools from practitioners [17]. This may indicate that the approaches are not aligned with practitioners' actual needs. However, improving these approaches has been hindered by the lack of empirical evidence regarding factors used for filtering and prioritizing code smells, i.e., there is no clear indication about what factors should be considered in each approach. One explanation for this situation is that empirical studies of code smells have focused mainly on the negative effects of code smells instead of how developers handle them. For these reasons, it is essential to identify the factors that should be considered, e.g., by gathering practitioners' opinions.

To start with, we performed a study on professional developers to determine the factors that they consider to handle code smells, especially during

the prefactoring phase. In this phase, developers refactor the source code before implementing their code [15]. As existing techniques can be classified into filtration and prioritization techniques, we consider both.

We conducted the study with ten professional developers by asking them to select and prioritize code smells of an open source project under the condition that they complete a list of five tasks. The reasons why they used for selecting and prioritizing code smells were then collected and analyzed. In total, we obtained 69 responses for code smell filtration and 50 responses for code smell prioritization.

The main contribution of this study is that we determined the factors considered by developers when filtering and prioritizing code smells for analysis in the prefactoring phase. This study may help researchers and tool developers focus on the most appropriate factors concerning code smells during prioritization and filtration which can improve the practicability and usability of code smell related tools. To the best of our knowledge, this is the first study to empirically investigate the factors considered by professional developers when filtering and prioritizing code smells.

The remainder of this chapter is organized as follows. First, we summarize related studies regarding empirical studies of code smells and techniques of code smell prioritization and filtration in Section 3.2. In Section 3.3, we explain the design and details of our study. We present the analysis of the results in Section 3.4. Threats to validity are discussed in Section 3.5. Then, we presented our future work in Section 3.6. Finally, we give our conclusions and suggest further studies in Section 3.7.

3.2 Related Work

3.2.1 Empirical Studies on Code Smells

Code smells are a very active topic for academic and industrial researchers throughout the world, and thus many empirical studies have been conducted to obtain insights into the issues that the community should explore. We outline empirical studies regarding code smells in this subsection.

Most previous studies have considered the negative effects of code smells. For example, Khomh et al. [48] and Guerrouj et al. [49] found that classes with code smells are more likely to change and become faulty. Abbes et al. [50] investigated the effects of code smells on program comprehension and found

that a combination of Blob Class and Spaghetti Code significantly decreased the performance of developers. Their results agreed with those obtained by Yamashita and Moonen [51], who found that inter-smell relationships were related to maintenance problems. In addition, Yamashita and Moonen [52] explained that code smells can be used to partly reflect maintainability aspects of software. Sjøberg et al. [53], however, found that 12 of the code smells in their experiment did not significantly increase maintenance effort. Nevertheless, Soh et al. [54] revisited the same dataset to conduct a deeper investigation and found that different code smells, in fact, impact the effort of different maintenance activities. For instance, searching effort is affected by Feature Envy while Editing effort is affected by Data Clumps code smells.

In addition to studies into the effects of code smells, previous studies over the past decade have investigated how developers deal with code smells. Yamashita and Moonen [17] reported a survey of 85 professional developers concerning code smells. Most of the subjects in their study stated the need for improved tools to detect code smells, especially tools with context-sensitive features. Palomba et al. [55] studied how developers perceive code smells and found that they perceived code smells differently as problems according to the types of code smells. Arcoverde et al. [56] performed an exploratory survey to understand why code smells remain in source code and found that one of the reasons was concern about breaking client code. Peters and Zaidman [57] investigated the lifespan of code smells by mining a software repository to determine the perspectives of developers regarding code smells. They found that developers were aware of code smells, but they were unlikely to solve them.

Previous empirical studies have mainly focused on the negative effects of code smells and how developers perceive them, but there is no empirical study that considers how developers select and prioritize code smells. This gap causes some difficulty in the study of the techniques regarding code smell filtration and prioritization due to a lack of a clear direction for the research community to focus on. Thus, the aim of this study was to address this shortcoming and identify the issues that the research community should consider by identifying the factors that are used by professional developers.

3.2.2 Code Smell Prioritization and Filtration

As the large number of code smells detected by detection tools is one of the crucial challenges, many studies have been conducted to prioritize and filter the code smell detection results. This subsection summarizes recent studies on code smell prioritization and filtration.

Severity, describing how strong a code smell is, is one of the most popular factors used for prioritizing code smells. For instance, Marinescu defined severity as "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold" [43]. Fontana et al. [41] also proposed the Intensity Index by calculating the distribution of software metrics. They represented the index as a numerical value between 1–10 and defined five intensity levels corresponding to each range: Very Low, Low, Mean, High, and Very High. This metric can be used to determine critical smells of the system.

Another approach that has been proposed to prioritize code smells is to use the context of developers. We define the Context Relevance Index by applying an impact analysis technique to a list of issues in an issue tracking system and prioritized code smells that were most likely to be related to the context of developers [44, 58]. The details can be found in Chapter 4.

Furthermore, many approaches have been proposed by using not only one, but multiple factors for code smell prioritization. Arcoverde et al. [45] presented an approach using four heuristics: change density, error density, anomaly density, and architecture role, to prioritize code smells. Vidal et al. [46] used three criteria: historical information of component modification, the relevance of the type of code smell from the developer's perspective, and modifiability scenarios of the system, to perform semi-automated prioritization of code smells. In Chapter 4, we also propose a prioritization technique using the combination of code smell severity and the Context Relevance Index.

In addition to code smell prioritization, many studies have been conducted to filter code smell detection results to increase the accuracy of existing tools. Fontana et al. [42] considered the application domain of the system to calculate strong and weak filters. They used a strong filter to remove false positive code smells from detection results and a weak filter to identify code smell instances that are not likely to be problematic. In contrast, Ratiu et al. [59] used historical information to measure the stability of each code

smell and filter out the instances that might not be harmful the system.

Although many approaches have been proposed, they use widely varying factors to prioritize and filter code smells. This may provide us with a hint that there is a lack of direction with respect to approach. In other words, researchers and tool developers do not actually realize the factors that should be considered in code smell filtration and prioritization. An underlying reason might be a lack of studies focusing on factors used by practitioners. However, in order to improve the usability and practicality of research tools in our field, the factors used by practitioners need to be considered when proposing new techniques. Therefore, the empirical evidence from this study should be able to provide an opportunity to support the advancement of such approaches by guiding the community to re-focus on the direction that practitioners need.

3.3 Study Design

As discussed earlier, the main purpose of this study is to identify the factors that practitioners use in the code smell filtration and prioritization process. The result of this study would allow researchers and tool developers improve code smell-related techniques such as filtration and prioritization by considering the use of the identified factors in the proposed techniques. As a result, the practicability and usability of research tools can be improved.

Many methods have been proposed based on the use of different factors to prioritize and filter code smells. Studies concluded that the main reason for refactoring by developers is to facilitate task implementation rather than removing the code smell itself [6, 9]. Therefore, in order to obtain empirical evidence of the factors used for filtering and prioritizing code smells, we conducted a study in a situation where many factors could be observed, including the factors related to the developer's tasks. Thus, we extended our controlled experiment in Chapter 4 on the code smells filtered by developers before working on specific tasks. While Chapter 4 focused only on the code smell filtration process, this work also analyzed the code smell prioritization process used by developers. In addition, we collected the reasons for their decisions and included the results of their analyses in this study. The details of our study are explained in the following.

3.3.1 Research Questions

To obtain empirical evidence of how developers filter and prioritize code smells, we focused on the following two research questions.

- **RQ3.1:** *What are the factors used by developers in the code smell filtration process?*
- **RQ3.2:** *What are the factors used by developers in the code smell prioritization process?*

In the first question, the filtration process is focused on code smells that developers need to address in a timely manner. The results could allow the research community to focus on factors used in practical code smell filtration process to reduce the number of false positives (code smells that are not harmful from the perspective of developers). The second question focuses on the prioritization process which defines the *order* in which code smells should be addressed based on the results of the code smell filtration process, which could facilitate more practical approaches to code smell prioritization by focusing on the key problems.

In order to answer both research questions, we aim to quantify the following three aspects.

1. **Factors used by developers:** As the main purpose of this study is to understand the factors that developers use to handle code smells, considering the number of occurrences of each factor would allow us to see the ranking and distribution of the factors used by the developers.
2. **Combination of multiple factors used by developers:** As discussed in Section 3.2, previously proposed techniques used not only one, but multiple factors to prioritize code smells. Therefore, considering the number of factors that are considered together would enable us to identify the practitioners' need and verify whether the research community's direction is aligned with it.
3. **Effect of developers' experience:** In the empirical study regarding software development, the experience of developers plays a particularly important role, e.g., developers with more experience tend to make a decision differently than developers with less experience. Thus, analyzing the result by classifying the experience of the subjects would

enable us to understand if there are any differences in the way more experienced developers handle code smells when compared to less experienced developers, i.e., there might be some factors that are considered by more experienced developers but not by less experienced developers and vice versa.

3.3.2 Data Collection

The data in this study were obtained from an extension of our study in Chapter 4. Our study in Chapter 4 deals with evaluating a code smell prioritization technique to determine whether it agreed with the process followed by professional developers. We employed the source code for the JabRef project[1], a list of five issues, gold set methods (methods modified to address each issue), and 22 code smells belonging to the Blob Class, Data Class, God Class, and Schizophrenic Class as detected by inFusion ver. 1.9.0 [30]. The subjects comprised ten professional developers with working experience ranging from 2–13 years. Each subject was provided with a list of five issues, including a summary and description, as typically shown on the issue tracking system for the task to be completed. In addition to the content for each issue, the subjects were provided with the solution for each issue in terms of the diff files in order to reduce the workload for the subjects. The subjects were then asked to consider all the issues and the related changes. Subsequently, a list of code smells, including the class name, package name, type of code smell, and a detailed description explaining why inFusion considered each problem to be a code smell were provided to the subjects. The source code for JabRef, including modules with code smells, was also provided. Finally, the subjects were requested to filter code smells that they considered should be refactored after considering all the information.

In Chapter 4, we focus only on the modules that developers considered should be refactored, whereas the main purpose of this chapter is to determine *why* they made these decisions. In addition, Chapter 4 focuses only on the code smell filtration process, whereas this chapter also investigates the code smell prioritization process. Therefore, we added the following experimental tasks for our subjects and performed further investigations by: 1) gathering more concrete evidence of why the subjects filtered or did not filter a specific code smell; and 2) asking the subjects to prioritize the code smells

[1] `http://www.jabref.org/`

TABLE 3.1: Examples of Responses and Corresponding Factors.

Response	Factors
It involves many issues.	Task relevance
This file has to be changed according to the issue list in this release. The class has too many functionalities and it is also hard to navigate.	Task relevance, Smell severity
[The related task is] not difficult to fix.	Task implementation cost
Util Class is a centric class. This class was invoked by many classes, so this class should be fixed first.	Module importance

that they filtered using a ranking scale (i.e., 1, 2, 3, ...) as well as their reasons. The responses obtained from the subjects allowed us to analyze the factors that affected their filtration and prioritization of code smells.

3.3.3 Data Analysis

After obtaining the results from the participants, we used a coding technique from grounded theory [60] to codify responses as *factors* because it is suitable for studying human aspects of software engineering [61]. In addition, this technique has been used widely in software engineering research, including studies of code smells [17]. The initial factors were generated first for the analysis. The factors were not fixed or limited, so they could be modified, added, or deleted if necessary. Two investigators completed the process by reading the responses of the subjects and assigning appropriate factors to each response. The investigators discussed the cases in the event of a disagreement of the results of the coding process. The factors used in this study were extracted by analyzing the responses of the subjects. So, some of them might be the same terms as the ones used in the literature while some of them might be different. Some examples of responses and their corresponding factors are given in Table 3.1. For instance, the response "*It involves many issues*" was assigned with the factor *Task relevance*, and the response "*This file has to be changed according to the issue list in this release. The class has too many functionalities and it is also hard to navigate.*" was assigned with the factors *Task*

TABLE 3.2: Final Factors.

Group	Factor	Description
Smell	Smell severity	The subject states whether the code smell is severe or not severe.
	Smell coupling	The selected smell is related to another smell and should be solved together.
	Co-located smells	Multiple code smells appear in the same module and should be solved at the same time.
Task	Task relevance	The smell either is related or not related to the subject's task , i.e., it appears in the module that the subject needs to modify to solve the task.
	Task importance	The related task is or is not important.
	Task implementation cost	The cost incurred for implementing a specific task is high or low.
	Task implementation risk	The risk of implementing a specific task is high or low.
Quality	Testability	The module is difficult to test, or the subject wants to improve the testability.
	Readability	The module is difficult to read, or the subject wants to improve the readability.
	Maintainability	The module is difficult to maintain, or the subject wants to improve the maintainability.
	Understandability	The module is difficult to understand, or the subject wants to improve the understandability.
Module	Module importance	The module plays an important role in the system, e.g., it is a main class that control business logic of the system.
	Module dependency	The module is dependent on another module and should be refactored in a specific order.
Refactoring	Refactoring cost	The cost incurred by performing refactoring operations to remove a code smell is high or low.
False positive	Smell false positive	The subject does not consider that the result obtained by the code smell detector is a code smell.

relevance and *Smell severity*. Moreover, factors such as *Smell false positive* were also assigned to the responses representing the reasons that the subject did not filter a specific code smell. At the end of the process, the investigators combined closely related factors and finally obtained 15 factors. The final 15 factors can be categorized into six groups based on the lexical and semantic characteristics of each factor, e.g., the factors containing the word *smell* that describe a characteristic of code smells were categorized into the group titled *Smell* and the factors containing the word *task* were categorized into group titled *Task*. However, the factors *Testability*, *Readability*, *Maintainability* and *Understandability* do not have a word in common but all of them are aspects of software quality. Therefore, they were categorized into a group titled *Quality*. The complete list and the explanation of each factor can be found in Table 3.2.

After obtaining the final factors, we have conducted follow-up interviews with three of the subjects to validate them. We provided the subjects' original responses, the factors and descriptions that we assigned in each response. Then, we asked them to confirm whether our factors and descriptions were aligned with their intention. All subjects in our follow-up studies confirmed and agreed with all factors that we assigned to their responses. In addition, because the factors identified might be specific to only the project used in this study, we have also asked the subjects whether the factors identified in this study were useful for filtering and prioritizing code smells in general. All of the subjects concurred.

To answer the first aspect (factors used by developers) and second aspect (combination of multiple factors used by developers) of each research question, we counted the number of factors and the number of each pair of factors that appeared together more than once respectively. As discussed earlier, one response may contain multiple factors as the subjects may have used multiple reasons to select or prioritize code smells. We then used the number of responses containing a specific factor to answer each research question. As for the third aspect (effect of developers' experience), we classified the subjects with less than five years' experience as junior developers and the subjects with more than or equal to five years' experience as senior developers. Specifically, the junior developer group includes developers with 2, 3, 4, 4, and 4 years of experience while the senior developer group includes developers with 5, 5, 12, 12, and 13 years of experience. As a consequence,

TABLE 3.3: Factors Used by Developers in the Code Smell Filtration Process

Factor	#All[1]	#Junior[2]	#Senior[3]
Task relevance	33	19	14
Smell severity	11	6	5
Task implementation cost	6	3	3
Testability	5	5	0
Co-located smells	4	1	3
Module importance	2	1	1
Readability	2	2	0
Smell false positive	2	1	1
Smell coupling	2	0	2
Maintainability	1	1	0
Understandability	1	1	0

[1] Number of responses from all subjects
[2] Number of responses from junior subjects
[3] Number of responses from senior subjects

TABLE 3.4: Combination of Multiple Factors Used by Developers in the Code Smell Filtration Process

Factor	Number of responses
Task relevance, Smell severity	9
Task relevance, Testability	5
Task relevance, Readability	2
Task relevance, Smell coupling	2

each group comprised of five developers equally. We did not use the position that the subjects specified at the beginning of the experiment because it might be difficult to compare among different companies. We then analyzed the number of factors mentioned by each group to see the differences.

3.4 Analysis of Results

3.4.1 RQ3.1: What are the factors used by developers in the code smell filtration process?

Factors used by developers

Table 3.3 shows the factors used by developers when filtering code smells according to our analysis. Column #*All* in the table represents the number of responses from all subjects of each factor. For example, there are 33 responses from the subjects containing content identified as *Task relevance* and 11 responses from the subjects containing content identified as *Smell severity* in the first and second row respectively. Each response represents the input from the subject when selecting and prioritizing a specific code smell. Clearly, *Task relevance* was the most common factor used in the filtration process, where developers tended to filter code smells related to their tasks. Some of the responses made by the subjects were very straightforward because they only considered the relevance to their tasks (e.g., "*It is related to the issue #4*"), whereas some of the responses were also concerned with factors in addition to *Task relevance* (e.g., the response "*It is related to an issue that we will address soon, and it would be good if we can separate the logic into another class to make it testable and more readable*" is concerned with the *Task relevance*, *Testability*, and *Readability*). These results support previous studies showing that developers tend to refactor source code mainly to support the implementation of their tasks [6, 9].

The second most common factor was *Smell severity*. We did not provide the subjects with the severity values obtained from the code smell detector used in this experiment in order to prevent cognitive bias, i.e., the subjects might have filtered code smells with high severity values without actually analyzing them. Instead, we provided the subjects with the source code related to each code smell for the analysis. When we analyzed the responses of the subjects, we assigned the *Smell severity* factor to responses that contained some specific adverbs related to a degree such as *too* or *very*, e.g., "*Functions have too many dependencies for example screens, menus, etc.*" This indicated that the developers tended to consider the *Smell severity* when they were filtering code smells.

In addition to the two factors mentioned above, the subjects also considered other factors. For instance, one subject stated the reason why they did not choose a particular smell as: "*This should be selected but we have just added a parameter in this release. The major change is in FieldContentSelector.java.*" This indicates that the developer also considered the *Task implementation cost*. Another subject indicated the reason why they filtered a code smell as "*Two code smells in one file,*" which demonstrates that factors such as *Co-located smells* were also considered by the subjects.

In conclusion, *Task relevance* and *Smell severity* were the most common factors used by developers in the code smell filtration process.

Combination of multiple factors used by developers

As mentioned earlier, previous studies often used multiple factors to detect and filter code smells, so we conducted further analysis on the factors considered together when developers filtered code smells, thereby obtaining insights into the factors that should be used together when detecting or filtering code smells. Thus, for each response given by the subjects, we counted each pair of factors that appeared together more than one time.

The results in Table 3.4 represent the pairs of factors that the subjects considered together when filtering a code smell, excluding the ones that were assigned to only one response. It is apparent that the *Task relevance* and *Smell severity* were the most common factors. For example, one of the subjects stated the reason why they filtered a code smell as: "*This file has to be changed according to the issue list in this release. The functions are too long,*" thereby mentioning two factors. The second item in Table 3.4 comprises the combination of *Task relevance* and *Testability*, where the results show that *Task relevance* was the most common factor considered by developers but it was also often considered together with another factor, i.e., *Testability* in this case (e.g., "*In this release, we have to add specific behavior to fix a bug in issue #4, and thus we have to refactor the code so that we can write the unit test more easily.*").

To sum up, *Task relevance* and *Smell severity* were the most common factors that were considered together by developers in the code smell filtration process.

Effect of developers' experience

As discussed earlier, since the experience of developers may affect the way they filter code smells, we also analyzed the factors classified by the years of experience of the subjects. The column #*Junior* and #*Senior* of Table 3.3 represents the number of responses from junior and senior subjects of each factor respectively. The result from our analysis showed that *Task relevance* and *Smell severity* were still the most popular factors even though we considered each group individually. Although there were factors that senior developers considered but did not appear in junior developers' responses (e.g., *Smell coupling*) and vice versa (e.g., *Testability*), we found that such differences were minor. We concluded that there is no significant difference between how junior and senior developers filter code smells.

To conclude, the experience of developers was unlikely to affect the way they filter code smells.

3.4.2 RQ3.2: What are the factors used by developers in the code smell prioritization process?

Factors used by developers

Table 3.5 shows the factors used by developers in the code smell prioritization process. The column #*All* in the table represents the number of responses from all subjects of each factor. In contrast to the factors used for code smell filtration, *Module importance* was the most common factor considered in the prioritization process. For instance, one of the subjects stated that they ranked a code smell with the highest priority because: "*Util Class is a centric class. This class is invoked by many classes. Thus, this class should be fixed first.*" Another subject stated that they ranked a code smell with the second highest priority because: "*It is important but less important than the main UI class.*" Thus, the developers tended to first prioritize modules with important roles in the system and then other modules with lower priority.

However, the second most common factor was still *Task relevance*. We found that the numbers of tasks related to code smells were often included in the responses. For example, one subject mentioned that they assigned a code smell as the first item to fix because: "*It involves many issues,*" and the reason for the second smell was: "*It only involves issue #1.*" Another subject gave a similar reason why a code smell was ranked first: "*This issue list has*

TABLE 3.5: Factors Used by Developers in the Code Smell Prioritization Process

Factor	#All[1]	#Junior[2]	#Senior[3]
Module importance	14	6	8
Task relevance	10	7	3
Testability	5	4	1
Smell severity	4	1	3
Maintainability	3	3	0
Refactoring cost	3	0	3
Co-located smells	2	1	1
Module dependency	2	0	2
Readability	2	2	0
Task implementation cost	2	2	0
Task importance	2	2	0
Task implementation risk	1	0	1

[1] Number of responses from all subjects
[2] Number of responses from junior subjects
[3] Number of responses from senior subjects

three issues related to this single file. This should be considered the highest priority to be fixed."

Furthermore, other factors such as the *Refactoring cost* (e.g., "*Low effort [for refactoring] is required.*") and *Module dependency* (e.g., "*This file should be addressed after the Util class to consider the lower risk of the code change.*") were also considered by the subjects.

In summary, *Module importance* was the most common factor used in code smell prioritization process and the second was *Task relevance*.

Combination of multiple factors used by developers

As discussed earlier, many prioritization techniques have been proposed based on combinations of multiple factors despite the lack of empirical evidence in support of this approach. To address this shortcoming, we also analyzed the combination of multiple factors used when developers prioritized code smells.

We asked the subjects to state their reasons for prioritizing each code smell, but we combined the factors for every response given by a subject

TABLE 3.6: Combination of Multiple Factors Used by Developers in the Code Smell Prioritization Process

Factor	Number of subjects
Module importance, Task relevance	4
Module importance, Testability	3
Task relevance, Testability	3
Co-located smells, Task relevance	2
Maintainability, Task relevance	2
Module importance, Readability	2
Module importance, Refactoring cost	2
Module importance, Smell severity	2
Readability, Task relevance	2
Readability, Testability	2
Smell severity, Task relevance	2

in the investigation, which differed from our analysis of the filtration process. This is because code smell prioritization is a comparative process, i.e., the subjects had to compare different smells and give higher importance to one smell, but less importance to others. Thus, we could determine the pairs of factors used by developers to prioritize code smells.

The results, excluding the ones that were assigned by only one subject, are presented in Table 3.6. According to these results, *Module importance* and *Task relevance* were the most common combination of multiple factors used by developers when prioritizing code smells (e.g., "*It should be fixed first because it is related to the issue and it is a share class.*"). In addition, the second pair that developers used most often was *Module importance* and *Testability* (e.g., "*It would be better if the main class of the UI project is readable and testable. In addition, it would reduce the time required for testing.*").

To conclude, *Module importance* and *Task relevance* were the most common factors that were considered together in the code smell prioritization process.

Effect of developers' experience

Similarly to RQ3.1, we analyzed the results by classifying according to junior and senior developers. Columns #*Junior* and #*Senior* of Table 3.5 show the

number of responses from junior and senior subjects of each factor, respectively. The results were similar to those in RQ3.1 in the sense that there was no significant difference between both groups. The most common factors for both groups were still *Module importance* and *Task relevance*. While there were some cases that junior and senior developers used different factors for prioritizing code smells, we found such differences insignificant and concluded that the factors used for prioritizing code smells are unlikely to be affected by developers' experience.

In conclusion, developers' experience did not tend to affect how they prioritize code smells.

3.4.3 Implications

Table 3.7 shows the results of our survey regarding the factors identified in this study that have been considered in the literature. The scope of our survey was limited to the code smell prioritization and filtration approach. Code smell prioritization uses the result of the code smell detection approach as one of the inputs. Then, it generates a list where code smells are sorted by specific criteria. However, code smell filtration reduces the number of code smells based on specific criteria. We put a checkmark in the cell where the factor in each row was used in the approach in each column. It is noteworthy that we compared the definition of the factor of each work with the one in this study. Therefore, the term used in the original paper may not be exactly the same as the one defined in this study. Furthermore, we list the factors that were used by work in the literature but were not identified in this study (e.g., *Change density*, *Error density*) in group *Others*.

The results showed that some of the factors identified in this study (*Smell severity*, *Co-located smells*, *Task relevance*, *Module importance*, and *Smell false positive*) were used in code smell prioritization and filtration approaches in the literature. This suggests that a part of the literature has used the factors that align with practitioners' needs. Therefore, we encourage the research community to continue focusing on the factors for future development, thus improving the practical utility of these research tools.

However, the results also show that existing techniques have paid most attention to factors in group *Smell*. In addition, out of the 15 factors identified in this study, ten of them have not been considered in the literature. This indicates that there are still many factors that need to be considered by the

TABLE 3.7: Factors identified in this study that have been considered in the literature

		Prioritization						**Filtration**	
Group	**Factor**	Marinescu [43]	Arcoverde et al. [45]	Fontana et al. [41]	Sae-Lim et al. [44]	Vidal et al. [46]	Sae-Lim et al. [47]	Ratiu et al. [59]	Fontana et al. [42]
Smell	Smell severity	✓		✓			✓		
	Smell coupling								
	Co-located smells		✓						
Task	Task relevance				✓	✓	✓		
	Task importance								
	Task implementation cost								
	Task implementation risk								
Quality	Testability								
	Readability								
	Mantainability								
	Understandability								
Module	Module importance		✓						
	Module dependency								
Refactoring	Refactoring cost								
False positive	Smell false positive								✓
Others[1]	Change density		✓			✓		✓	
	Error density		✓						
	Smell type					✓			
	Smell persistency							✓	

[1] Other factors that were not identified in this study.

research community. For instance, even though factors in group *Task* have been considered in recent studies, the only factor that has been considered is *Task relevance*. In contrast, the factor *Task implementation cost* has never been considered although it was the third most common factor in the code smell filtration process identified in this study. In addition, factors in group *Quality* and *Refactoring* have never been taken into account even though they are closely related to code smells. We strongly recommend considering these factors when developing new techniques such as code smell filtration and prioritization.

Lastly, although there are some techniques that have proposed the combination of multiple factors when prioritizing code smells, there is still room for improvement in this aspect. For example, the combination of *Module importance* and *Task relevance* has never been proposed in spite of being the most common used in the code smell prioritization process. Therefore, we encourage the development of new techniques that consider the combinations observed from this study.

3.5 Threats to Validity

The internal validity of this study is dependent on the data set that we used. First, the data set was designed for evaluating a code smell prioritization technique for prefactoring, so it may have missed some factors considered by developers in different situations. For example, our results in Table 3.7 shows that some existing techniques used factors such as *Change density* or *Error density* to prioritize or filter code smells. However, because this study did not include historical information, e.g., version control system or issue tracking system, we may have failed to identify such factors that rely on historical information. Second, the subjects were not the main developers of the project investigated in this study. Therefore, the results may differ if experiments are conducted on a source code that the developers are familiar with; for instance, they may consider some code smells as false positives or may filter code smells that are indirectly impacted by changes. However, conducting this type of study with the project that the subjects are working on is not practical due to the fact that we need to access to their source code and issue tracking system. Additionally, conducting a study regarding code smells

with industrial developers using open source software source code is not uncommon (e.g., [55]). Inviting core developers of open source projects could become a part of our future work, but a low response rate is expected [55].

The construct validity might depend on the factors that we assigned to each response because there may have been some bias during the process. Furthermore, the responses obtained from the subjects might not represent the actual reasons why they filtered or prioritized code smells. However, all subjects in our follow-up studies confirmed and agreed with all factors that we assigned to their responses. While only 10 factors were in this follow-up study, which does not cover all the factors identified in this study, we believe this threat should be minimized.

Another construct validity might depend on the validity of the answer that the subjects made, i.e., the subjects might have randomly selected or prioritized code smells without consideration. However, as our study design forced the subjects to state the reasons for the performed action, i.e., the reasons for selecting or prioritizing particular code smells, we believe this threat is mitigated. In addition, because we mainly focused on the reasons behind the selected or prioritize code smells in this study, the validity should depend on the reasons that the subjects state and not the actions that they performed, i.e., if the subjects did not state the reasons, the result will be invalid. However, such cases did not occur during our study.

Finally, similar to other empirical studies, the external validity of this study is dependent on the scale of the experiment. The subjects differed in terms of their background and the number of years of experience. However, the number of developers who participated in this study was only ten, and they might not have been representative subjects. In addition, this study only investigated one project with four types of code smells. Therefore, the factors identified in this study might be specific to only this project. However, in our follow-up studies, all of the subjects agreed that the factors identified in this study were useful for filtering and prioritizing code smells in general. This may indicate that the identified factors are not specific to the project used in this study. Nevertheless, a larger scale study might be beneficial.

3.6 Future Work

Our future work includes two parts. As discussed earlier, because we may have missed some details in this study, the first task is to conduct follow-up interviews with some of the subjects to obtain a more specific opinion on how they filter and prioritize smells. The second part of our future work is to propose code smell filtration and prioritization techniques based on the factors identified in this study that have not been considered by the research community as discussed in Section 3.4.3.

3.7 Conclusion

In this study, we conducted an experiment to determine how professional developers filtered and prioritized code smells. Our findings agree with previous studies where developers gave higher priority to code smells related to their specific context, i.e., the tasks upon which they were working. First, *Task relevance* was the most common factor considered during code smell filtration, followed by *Smell severity*, and both were also often considered together during the filtration process. Second, *Module importance* was the most common factor in the code smell prioritization process, followed by *Task relevance*. Moreover, other factors such as the *Task implementation cost* and *Co-located smells* were considered in both processes. We recommend that researchers and tool developers focus on the factors used for code smell filtration and prioritization identified in this study.

Chapter 4

Context-Based Code Smells Prioritization

Summary

Existing techniques for detecting code smells (indicators of source code problems) do not consider the current context, which renders them unsuitable for developers who have a specific context, such as modules within their focus. Consequently, the developers must spend much time identifying relevant smells. We propose a technique to prioritize code smells using the developers' context. Explicit data of the context are obtained using a list of issues extracted from an issue tracking system. We applied an impact analysis to the list of issues and used the results to specify the context-relevant smells. Results show that our approach can provide developers with a list of prioritized code smells related to their current context. We conducted several empirical studies to investigate the characteristics of our technique and factors that might affect the ranking quality. Additionally, we conducted a controlled experiment with professional developers to evaluate our technique. The results demonstrate the effectiveness of our technique.

This chapter is based on our journal paper "Context-Based Approach to Prioritize Code Smells for Prefactoring" [47] published in Journal of Software: Evolution and Process, Copyright © 2017 John Wiley & Sons, Inc. The paper is an extended version of our earlier work "Context-Based Code Smells Prioritization for Prefactoring" [44] appearing in the 24th IEEE International Conference on Program Comprehension (ICPC'16), Copyright © 2016 IEEE.

4.1 Introduction

Code smell is an indicator of design flaws or problems in the source code [1, 2, 3]. Code smells of many types are summarized as smell catalogs [1, 2, 3]. Code elements that are affected by code smells are recommended for improved quality because they are likely to cause difficulties in the future, as discussed in the literature, e.g., making the source code more difficult to understand and maintain, or increasing code components' fault-proneness. Moreover, Yamashita et al. [19] found that code smells can affect important maintainability aspects. Hermans and Aivaloglou [20] conducted a controlled experiment, which revealed that code smells influence the block-based programming performance of novice programmers.

Many researchers have examined factors that can introduce code smells into the system. Tufano et al. [21] conducted a large-scale empirical study and found that the main activities which tend to introduce code smells are implementing features and enhancing existing ones. Moreover, they found that developers with high workloads and pressure are more likely to introduce smelly code to the system. Vale et al. [22] investigated the factors that can influence code smells based on a software developer's perspective. That investigation revealed that factors such as time pressure, bad design decisions, lack of business knowledge, inexperienced developers, little time for refactoring, and low priority assigned to software quality influence the severity of smelly code.

For an issue-driven software development project, development teams tend to adopt an issue tracking system such as Jira or Bugzilla to manage their lists of issues, so-called *backlogs*. Such issues can be anything related to software change requests, e.g., bug fixing or feature implementation. Developers apply refactoring techniques to source code before implementing changes according to issues during the so-called *prefactoring* phase [15]. Given those assessments, modules that are relevant to developers' tasks are likely to have higher priority for fixing of code smells and improvement of code quality over others because fixing their code smells might contribute to the support of future implementation, for instance, by improving the understandability and extensibility of the source code. Consequently, such modules are suitable to be regarded as their context. As described in this chapter, we use the definition of *context* similarly to *task context* defined by Kersten and Murphy as "the information–a graph of elements and relationships of program

artifacts–that a programmer needs to know to complete that task" [10]. Because the tasks are described as issues and are managed in an issue tracking system, we use the information in the issue to estimate the context.

Nevertheless, most existing techniques for detecting code smells and recommending refactoring opportunities ignore such a context and output the results only from analyzing the whole source code without prioritizing the results or prioritizing them with factors that are unrelated to the developer context. Therefore, the time-consuming process of identifying relevant outputs is left to developers, which is similar to the fact that static analysis tools are not used well by developers because of numerous detected warnings [14]. One example is the severity-based prioritization which rank first the most severe code smells [41, 42, 43]. This scheme is widely adopted when a developer wants to improve the overall system quality because solving code smells that contribute most to the decay of the source code is likely to be more useful than solving those which have little effect on the system. However, presuming the prefactoring phase, the developer has the specific context that the developer must work on. In this situation, solving the most severe code smell might not be the best strategy because solving such smells might not contribute directly to support of developers' implementation. In contrast, solving code smells according to context-based detection or prioritization might provide better alternatives because solving context-related smells is likely to facilitate the implementation of developers. Furthermore, in a real world situation, because development teams have limited time for delivering the product, and very little for improving the source code quality, it would be challenging to have permission from management for improving the overall software quality. However, in the prefactoring phase, the developer can use a part of the normal implementation time to improve the quality of only those modules that the developer is going to work on. This chapter therefore particularly addresses the importance of context-based prioritization rather than severity-based prioritization to reflect the real world situation in which the developer would have a limited amount of time to improve software quality.

Motivation. A clear necessity for a context-aware approach to code smell detection is apparent in recent research conducted in this area. Work done by Yamashita et al. [17] seems to show that developers need code smell detectors that support context-sensitive or domain-specific strategies. Work done by Bavota et al. [9] suggests that the developer's perspective should be considered when proposing refactoring recommendations. Many studies have

emphasized the importance of this prefactoring phase. Meng et al. [8] reported that developers do not typically refactor their code unless they must change the code to fix some bugs or introduce new features. Bavota et al. [9] supported this statement by asserting the possibility that the only developer goal of refactoring is to prepare source code for future changes. In addition, Silva et al. [6] conducted a large-scale study on the motivations underlying refactoring. Their results show that developers perform refactoring activity based mainly on change requirements. In other words, developers do refactoring to facilitate the maintenance task that they are working on. Based on their study, they also suggested that research on refactoring recommendation should specifically examine maintenance-task-oriented rather than code-smell-oriented solutions. In maintenance-tasks, developers specifically examine modules to be maintained, i.e., contexts for the maintenance task. Their work also suggests the motivation of context-based approaches.

Proposed Solution. We propose a technique to prioritize code smells from code smell detectors by considering developers' current context. This technique specifically examines support of the *prefactoring* phase [15]. During the prefactoring phase, developers facilitate implementation by improving source code extensibility and understandability by refactoring the source code before implementing features. Using the proposed technique, change information in an issue tracking system that is to be implemented using a particular milestone are used to estimate the developers' context. Such changes are to be implemented by modifying or extending existing modules in the source code. These existing modules can be located using impact analysis [15, 16]. We regard such relevant modules as their context because they are likely to be the location for implementation of new features in the change information. As a result, code smells that appear in relevant modules should be assigned higher priority so that the developers can readily distinguish them from irrelevant code smells. With support from our technique, developers can obtain a prioritized list of code smells based on their relevance to the context.

Evaluation Scheme. We conducted empirical studies to assess characteristics of our technique as well as factors that might affect the results obtained using our technique. We also conducted a controlled experiment with professional developers to evaluate our technique. The results demonstrate the effectiveness of our technique.

Contribution. The main contributions of this chapter are the following.

1. We propose a technique for prioritizing code smells using developers' context.
2. We define Context Relevance Index (CRI) based on automated impact analysis to use as a criterion to prioritize code smells.
3. We present empirical studies of the characteristics of this approach and factors that can affect the quality of the ranking results.
4. We present a controlled experiment showing that our technique can prioritize first code smells chosen to be prioritized by professional developers.

Chapter Structure. The remainder of this chapter is organized as described below. The next section presents our approach and its implementation. Section 4.3 presents research questions of this chapter. Our empirical studies using existing datasets are presented in Section 4.4. We then show our controlled experiment with professional developers in Section 4.5. We summarize related work in Section 4.6. Finally, Section 4.7 concludes this chapter and provides additional directions of research that should be explored.

4.2 Proposed Technique

4.2.1 Framework for Context-Based Code Smells Prioritization

We propose a technique for prioritizing code smell detection results from existing code smell detectors by considering the list of issues in the issue tracking system which developers must solve. Figure 4.1 presents an overview of our technique. Each gray node represents a subprocess of our technique. The input of the process is a list of change information obtained from the issue tracking system, the source code of the targeted project, and additional information for impact analysis such as the software repository. The output is a prioritized list of code smells based on their relevance to the developer context. Our approach first uses an impact analysis technique to obtain a list of modules that are likely to be the targeted modules of each change information. Next, we generate a list of code smells by application of an existing code smell detector with the target project source code. Then, for each code

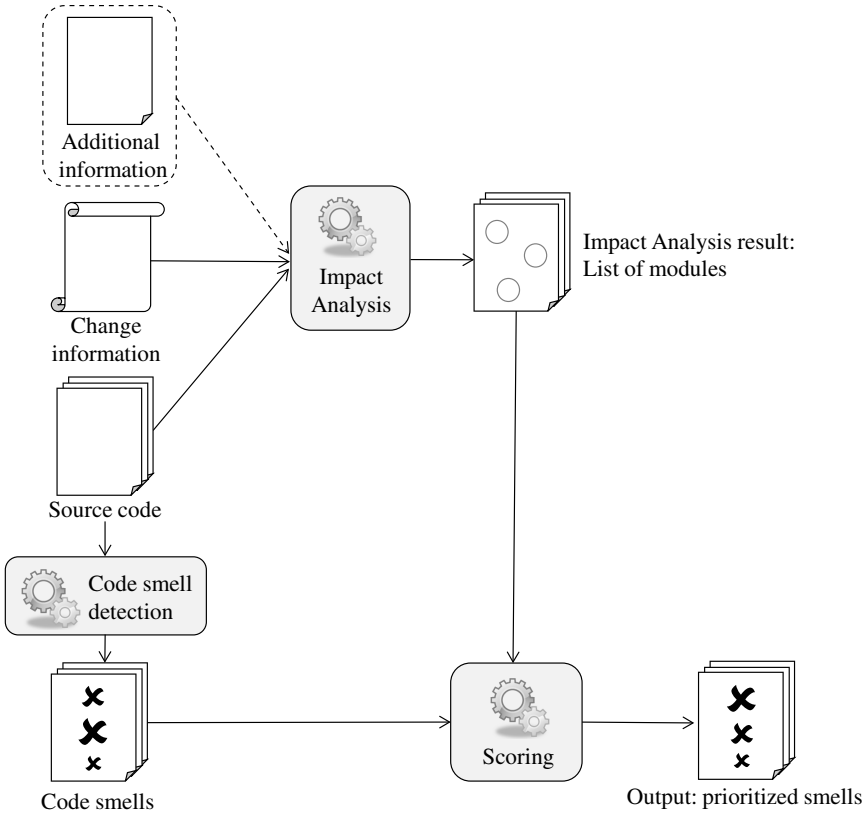


FIGURE 4.1: Overview of the proposed technique.

smell in the list, we calculate the *Context Relevance Index* (*CRI*) based on the relevance of the list of modules from impact analysis. Finally, we output the prioritized list of code smells ordered by the *CRI* value.

We design our approach as a framework within which code smell detection and impact analysis are its hot spots. Our prioritization technique works by connection with existing impact analyses and code smell detection techniques, and generates the results. In this sense, this framework is general and is independent of specific types of code smell detection technique or impact analysis. The following subsections present an explanation of these techniques and their application.

Impact Analysis

During software development processes, developers can modify some modules to satisfy a change request, such as one for bug fixing or feature implementation. Developers might first use their experience, system knowledge,

Summary:	Autosave turned off for Untitled Documents.
Description:	I have an issue where sometimes untitled documents get left on to disk. I think it's an issue when the app closes improperly in some way. An option to never save untitled documents would be a nice feature and would solve this issue.

FIGURE 4.2: Portion of an issue in the issue tracking system.

or technique such as feature location [16] to identify at least one module that is relevant to the change request. Then, they perform impact analysis to specify the full impact set, where the system is likely to be affected by such changes [15, 62].

Impact analyses of various types require different inputs such as natural language query, execution scenarios, and source code artifacts [16]. Gethers et al. [33] proposed a combination technique to perform impact analysis depending on the source of contextual information available to each software project, such as information retrieval, mining software repository, or dynamic analysis.

Figure 4.2 presents an example of an issue in issue tracking system of jEdit[1] project. We consider the information in *Summary* together with *Description* field as change information because they contain important details related to the change such as problems, reproduction steps, or even solutions. Therefore, this information is likely to be useful with impact analysis to identify source code components related to developers' context. We exclude comment fields in our approach because it is not always available at the beginning of each release. They might be generated in later stages of software development such as when a developer starts working on the issue and need clarification. Change information can be extracted using API of each issue tracking system or text mining tools such as Kurya by Kohan et al. [63], which is tailored specifically for mining issue tracking systems. Their tool can extract up to 3.5 documents per second at the network speed of 50 Mbps for responsive resources.

[1] `http://www.jedit.org/`

TABLE 4.1: Portion of code smell detector results.

Type	Entity	Granularity	Severity
SAP Breakers	`org/gjt/sp/jedit`	Subsystem	6
Blob Class	`org.gjt.sp.jedit.Buffer`	Class	7
Feature Envy	`buildDOM() : void`	Method	10

In our technique, we chose impact analyses that take change information i and source code C as inputs and provide a set of modules $M = \{\ldots, m, \ldots\}$ with their probability score as outputs because we specifically examine support of issue-based software development projects in which developers tend to implement features or fix bugs by following the change information in an issue tracking system. Therefore, we can use the change information as inputs of impact analysis to identify a set of modules that are likely to be modified to achieve the change. Consequently, applying the prefactoring technique to these modules is likely to support the developers' implementation by improving the understandability or extensibility of the source code.

As described in this chapter, we input a set of change information $I = \{i_1, \ldots, i_n\}$ and source code C to the impact analysis. A set of change information I represents a fixed list of issues that developers need to be implemented for a specific milestone. In this technique, we do not consider the situation where a list of issues is not fixed, e.g., more issues are added to the list after the prefactoring phase. Then, we obtain a series of sets of modules $\{M_1, \ldots, M_n\}$ together with its score indicating the relevance between the change information and the module. As described herein, modules can be either classes or methods.

Code Smell Detection

Code smell detection generates a list of code smells from the targeted source code. One example of the approaches is to detect them based on particular metric values such as lines of code (LOC). The input is the source code that we wish to analyze. The output is a list of smells. Each smell consists of attributes $\langle type, entity, granularity, severity \rangle$, where *type* stands for the type of the detected smell, *entity* signifies the module having the detected smell, *granularity* denotes the level of code smell consisting of the subsystem, class, and method (details of the difference between method level and class level

code smells will be discussed in Section 4.4), and *severity* is an integer value representing the smell strength, for example, Marinescu defined as "Severities are computed by measuring how many times the value of a chosen metric exceeds a given threshold" [43]. Table 4.1 presents an example of the code smell detection result from inFusion. For example, the second row shows the Blob Class smell in `Buffer` class with severity 7. We have omitted the package and class name of the method level smell.

In this approach, we apply a code smell detector and obtain the list of smells S.

Scoring

To assign the priority of each smell, we define the *Context Relevance Index* (*CRI*) attribute. The value of the CRI attribute is calculated using the weighted summation of the number of the modules in the result from impact analysis that match each smell's entity. The CRI of each $s \in S$ is definable as

$$CRI(s) = \sum_{i=1}^{n} \sum_{m \in M_i} \begin{cases} w(m), & \text{if } match(m, entity_s) \\ 0, & \text{otherwise,} \end{cases} \tag{4.1}$$

where n stands for the fixed number of considered issues (i.e., the number of issues that developers need to solve for the next milestone). $entity_s$ signifies the module having the detected smell s which can be a method, a class, or a subsystem. The predicate $match(m, entity)$ holds when module m is the same as or belongs to *entity* of smells. For example, if m is a method, and *entity* having a code smell is also a method, then $match(m, entity)$ holds when m and *entity* are the same. For the case in which m is a method, but *entity* having a code smell is a class, $match(m, entity)$ holds when m is in *entity* class. Finally, $w(m)$ stands for the probability score of a module which is a result of impact analysis.

4.2.2 Implementing Our Approach

We have implemented an automated tool[1] for use with the proposed technique. The chain is designed to connect with an existing impact analysis tool. When executed, our tool reads a file containing a list of code smells that

[1] `https://github.com/salab/CodeSmellsPrioritizer`

were generated by a code smell detector and calculates the CRI of each smell based on the relevance of the result from impact analysis.

Impact Analysis

We consider an impact analysis as a hot spot of the framework. Therefore, it is not limited to any specific technique. As an implementation in this chapter, we follow approaches described in work by Gethers et al. [33]. Their work was undertaken to integrate different impact analyses to improve the overall accuracy of the respective independent techniques, but also showed that different impact analysis approaches yield different accuracies of the results. They discussed differences among combinations of these three techniques: information retrieval (IR), dynamic analysis (Dyn), and mining software repository (MSR). The next few paragraphs present overviews of the respective techniques we used.

Information Retrieval (*IR*), such as Vector Space Model (VSM), is done by representing documents and queries as vectors [31]. Then, the relevance of documents and queries is obtainable by calculating their mutual cosine similarity. For impact analysis, documents are source codes of a specific project. Queries are change requests from the issue tracking system. Results then show the relevance between the source code and change requests. As described in this chapter, we follow Gethers et al.[33] using Latent Semantic Indexing (LSI) [32] for IR-based impact analysis. Actually, LSI is based on VSM. It was created with the intention of handling the situation of synonymy, a group of words that share similar meanings, and polysemy, words that have multiple meanings. Therefore, LSI is used for assessment of the similarity between documents from the common words that two documents contain rather than simple terms. The more words they have in common, the more similar they are. Details of the techniques we used can be found in an earlier report of the literature [33].

Dynamic Analysis (*Dyn*) uses the execution traces of the given program. Several technologies are available to collect the information, e.g., Java Platform Debugger Architecture (JPDA)[1] or Test and Performance Tools Platform (TPTP)[2]. In some cases, the execution trace of the specified change request is attached by the submitter. However, developers themselves can obtain it by

[1] `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/`
[2] `https://projects.eclipse.org/projects/tptp.platform/`

reproducing the steps specified in the change request as well. Such run-time information is useful to filter out the results from IR technique because the modules that are not in the execution trace are unlikely to be affected by the change request. Therefore, it is useful as additional information in our technique.

Mining Software Repository (*MSR*) is an approach to apply data mining techniques to software repositories such as version control systems. We specifically examine this technique on mining evolutionary coupling, i.e., modules that tend to be modified together. Such a technique can be done using association mining, i.e., the probability of *consequent Y* appears when *antecedent X* appears ($X \Rightarrow Y$) is represented by *confidence* and *support* values. Practically speaking, when a developer does association mining, the developer will select a module to use as the starting point using the developer's own system knowledge or techniques such as feature location. Then, the MSR-based impact analyses will generate a list of modules that are often changed together with the starting point module. Tools such as ImpactMiner [64] are useful to use a software repository as additional information in our technique. The following paragraphs explain how we combined each technique.

Information Retrieval and Dynamic Analysis (*IR+Dyn*) first use the IR technique to obtain a list of modules that are likely to be related the specified change information. Then, we use the result from Dyn to filter out the result from IR technique. We remove the method in results from IR if it is not in the result from Dyn because Dyn recorded the modules that have been executed during the operations. Therefore, modules that were not executed during the operations are unlikely to be related with the changes.

Information Retrieval and Mining Software Repository (*IR+MSR*) can be done by first obtaining the list from each technique, as described earlier. Then, one must alternately merge the results from each list until the lists are exhausted. If the same module appears in both lists, then it will be selected only once, but the score value will be the accumulation from both lists.

Information Retrieval, Dynamic Analysis, and Mining Software Repository (*IR+Dyn+MSR*) can alleviate the situation in which Dyn filters out the correct result from IR list [33]. Using the same heuristic as IR+MSR, this combination can be acquired by merging the results from IR+Dyn and MSR.

In summary, the techniques that we used are information retrieval (IR), the combination of information retrieval and dynamic analysis (IR+Dyn), the combination of information retrieval and mining software repository (IR+MSR),

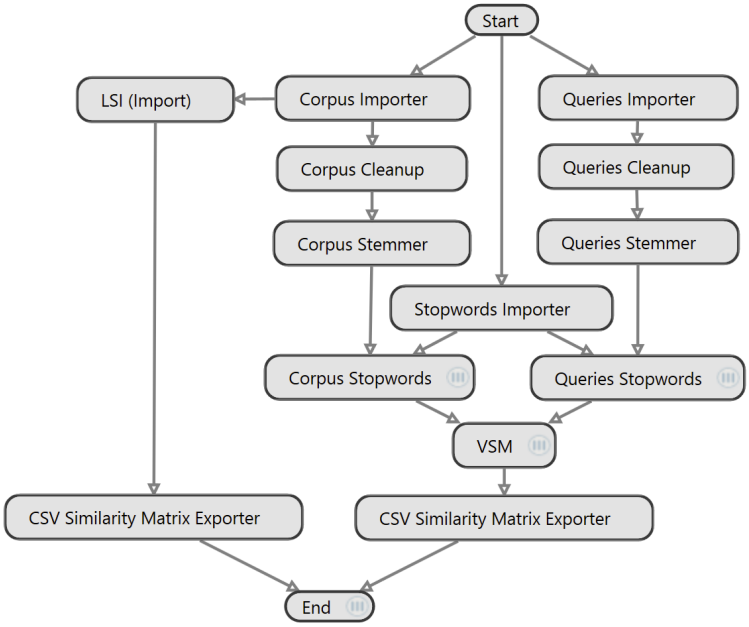


FIGURE 4.3: TraceLab configuration.

and the combination of information retrieval, dynamic analysis, and mining software repository (IR+Dyn+MSR). For IR and IR+Dyn, we use tools proposed by Dit et al. [65] together with a TraceLab-based solution [66, 67] to compute latent semantic indexing for information retrieval impact analysis. Then, we filter out the unrelated modules using the execution trace available in the dataset, as explained later. Figure 4.3 portrays our configuration in TraceLab, which is based on [67]. For IR+MSR and IR+Dyn+MSR, we use ImpactMiner [64] to perform association mining for MSR part. Then, we combine the results with IR and IR+Dyn as described earlier.

Code Smell Detection

As discussed earlier, because we designed the approach as a framework, it is independent of specific code smell detection. As an implementation in this chapter, for this study, we used inFusion ver. 1.9.0 [30] as a code smell detector because 1) it can detect code smells of 24 types, e.g., Blob Class, Data Class, or Feature Envy, 2) all detected smells are associated with the severity score, and 3) its detection process can be assembled in an automated manner. These characteristics are suited to our approach.

TABLE 4.2: Target code smells detected by inFusion.

Granularity	Type
Class	Blob Class [1, 3, 68]
	Data Class [1, 3, 69]
	Distorted Hierarchy [69]
	God Class [1, 3, 69]
	Refused Parent Bequest [3, 69, 70]
	Schizophrenic Class [69, 70]
	Tradition Breakers [3, 69]
Method	Blob Operation [1, 3, 68]
	Data Clumps [1]
	External Duplication [1, 68, 71]
	Feature Envy [1, 3, 69]
	Intensive Coupling [1, 3, 69]
	Internal Duplication [1, 68, 71]
	Message Chains [1, 3, 68]
	Shotgun Surgery [1, 3]
	Sibling Duplication [1, 68, 71]

A list of code smells detected by inFusion that are considered in this chapter, i.e., the smells of class and method levels, together with each one's related literature are presented in Table 4.2.

Scoring

Figure 4.4 represents an example of CRI calculating mechanism of ArgoUML project using IR+Dyn approach as an impact analysis. In this case, the God Class code smell in `Project` class obtains the CRI value as high as 4.39 because the `Project` class appears on the top position in many results from impact analysis. For example, the impact analysis using the change information of the issue #2500 predicts `Project` class with the probability 0.19, which is the third highest. On the other hand, the Blob Class code smell in `GeneratorCSharp` class obtains only 0.48 of CRI value because the class does not frequently appear in the top position of the results from impact analysis.

Using CRI to prioritize code smells instead of severity is likely to change the list characteristics. For instance, considering Fig. 4.4, it is apparent that the Blob Class smell in `GeneratorCSharp` class which has severity of 8 is assigned the lower position on the context-based prioritized list. On the other

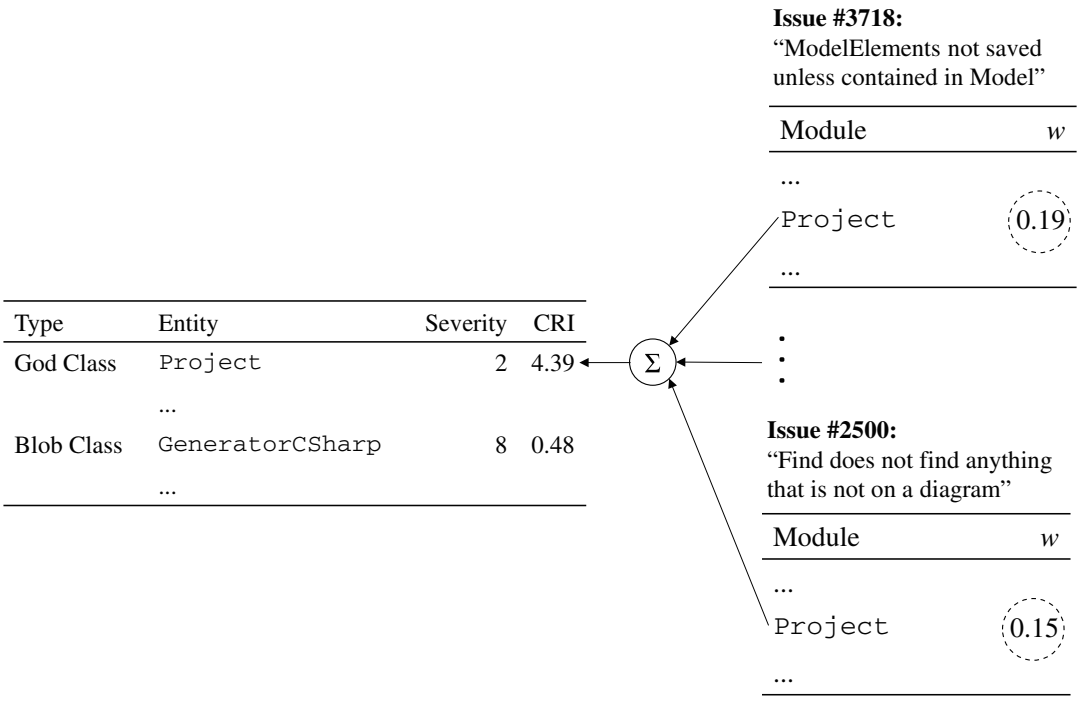


FIGURE 4.4: Example of CRI calculating mechanism.

hand, if one considers code smell God Class of class `Project` with severity 2, then this smell is assigned to the highest of the result in the context-based prioritized list, which indicates that even though it is not severe, our technique predicts that this smell is related to many issues associated with the issue tracking system of developers.

Therefore, when one prioritizes code smells using CRI, the most severe code smells might become least important. The least severe code smells might become most important depending on their relevance to the context of developers. The underlying reason is our assumption that solving smells with low severity but high relevance to the developers' context is better than solving smells with high severity but not relevant to the developers' context. In the prefactoring phase, solving smells in the modules that developers are unlikely to touch is not going to support their implementation. However, solving smells for modules in which developers are likely to implement features or fix bugs, even though the code smells are not very severe, would facilitate their implementation by improving several factors such as understandability and extensibility. Details of the difference between severity-based and context-based prioritization will be discussed in Section 4.4.

4.3 Research Questions

The main objectives of the studies described in this chapter are twofold: 1) elucidating the characteristics of our approach and the factors that might influence the ranking results and 2) evaluating whether our approach can prioritize code smells that align with developers' needs in the prefactoring phase. We divide the research questions into empirical studies and controlled experiment. The next subsections explain the details together with their associated research questions.

4.3.1 Empirical Studies

In this chapter, we divided the studies into two parts: 1) investigating factors affecting the performance of our approach and 2) studying the meaningfulness and usefulness of the recommended prioritization. The first part is intended to study the factors that can affect our approach whereas the second part is designed to validate our approach and to analyze different opportunities to apply our technique. Particularly we addressed the following research questions in each part.

Investigating Factors Affecting the Performance of Our Approach

- **RQ4.1:** *Which code smell granularity is more appropriate for our technique: Coarse-grained or fine-grained?* Code smell detection tools often classify smell granularity as coarse-grained or fine-grained, i.e., class level and method level. Solving code smells of both types is likely to improve the source code quality. Solving code smells in the module that a developer intends to modify would support its implementation. In the case of coarse-grained smells, solving the code smell in one class is likely to support the implementation under that class, including implementation of the methods of that class as well. However, in the case of fine-grained smells, solving the code smell in one method is likely to help only the implementation under that method, even though it is easier to understand how to solve the code smell. Therefore, we suspect that, in context-based prioritization, coarse-grained and fine-grained granularity code smells are expected to yield different ranking quality.

- **RQ4.2:** *Does the impact analysis accuracy affect the ranking quality?* As described in this chapter, our proposed technique was designed to use existing impact analyses. However, as we discussed, impact analyses of many types have been proposed in the literature. Different techniques might use inputs or mechanisms of different types. Nevertheless, the main objective is to achieve accuracy that is as high as possible because accuracy is an important factor of impact analysis. Therefore, understanding how different impact analyses affect the ranking quality can enable us to find the appropriate impact analysis for context-based code smell prioritization.

Studying the Meaningfulness and Usefulness of the Recommended Prioritization

- **RQ4.3:** *Does context-based smell prioritization provide more relevant results than severity-based smell prioritization?* The main objective of this study is to propose that context-based smell prioritization can be an efficient method for support of the prefactoring phase. Therefore, we investigate this research question to ascertain whether context-based smell prioritization produces more suitable results for supporting the prefactoring phase than the severity-based approach.

- **RQ4.4:** *What is the effect of the combination of severity-based and context-based prioritization?* In RQ4.3, we use context-based prioritization using the *CRI* value, with the expectation that code smells that are more related to the context would be at the top of the list. That is to say, the only factor that concerns the prioritization scheme is relevance to the context. However, the *severity* value, an extremely popular factor for prioritization, as we discussed, is also important information related to each smell. Therefore, it would be useful if we were able to use not only the *CRI* but also the *severity* to prioritize code smells to acquire the result that code smells with high relevance to the context *and high severity* are at the top of the list.

- **RQ4.5:** *Can our approach predict the smells to be refactored?* Although the aim of our technique is to *prioritize* code smells that developers should solve, one might have a question as to whether our technique can *predict* the smells that are going to be refactored by a developer, although

such is not our intention. To answer this question, we conducted an empirical study to ascertain whether the provided ranking of smells by the proposed technique fits the modules to be refactored.

4.3.2 Controlled Experiment

The main goal of our approach is to prioritize code smells that can support a developer's prefactoring process. Therefore, we conducted a controlled experiment with professional developers with the aim of answering the following research questions:

- **RQ4.6:** *How does our technique prioritize code smells selected by professional developers in the prefactoring phase compared to a severity-based approach?* Because the objective of our approach is to put first code smells that developers should be solved during the prefactoring phase, it is important to evaluate whether the result of our approach is in accord with the developers' opinion.
- **RQ4.7:** *What is the most appropriate proportion of linear combination of severity-based and context-based prioritization for the prefactoring phase?* In RQ4.4, we study the usage of the linear combination between context-based and severity-based approaches. However, in this study, we want to find out the most appropriate proportion of context-based and severity-based approach in the prefactoring phase.

4.4 Empirical Studies

4.4.1 Experimental Setup

Data Collection

In this evaluation, four open source projects, ArgoUML[1], Jabref[2], jEdit[3], and muCommander[4], were our subjects because their data are available through the benchmark dataset of Dit et al. [16]. The dataset includes lists of *Summary* and *Description* information, a list of executed methods, and gold set

[1] `http://argouml.tigris.org/`
[2] `http://www.jabref.org/`
[3] `http://www.jedit.org/`
[4] `http://www.mucommander.com/`

TABLE 4.3: Dataset Information.

Project	Version	Size (LOC)	# Issues	# Smells (M)[a]	# Smells (C)[b]
ArgoUML	0.22–0.24	309,468	91	412	61
JabRef	2.0–2.6	72,555	39	60	37
jEdit	4.2–4.3	140,592	150	184	51
muCommander	0.8.0–0.8.5	88,455	92	44	41

[a]Method level
[b]Class level

methods, and methods that were actually modified to solve extractable issues in issue tracking system, of each issue during the analyzed period. Table 4.3 presents information of our datasets including the size of the source code of the earlier version, the number of issues that we used between the two releases, and the number of smells detected by inFusion. For this study, we mined over 6,000 commits between releases 0.14–0.22 of ArgoUML, over 1,400 commits before release 2.0 of JabRef, over 900 commits between release 4.0–4.2 of jEdit, and over 1,600 commits before release 0.8.0 of muCommander to conduct MSR-based impact analyses. We assumed a situation in which developers are working on upcoming releases where the modules that they must work on are regarded as their context. Consequently, the issues describing developers' task for the upcoming releases are used to estimate the context.

We first defined the oracle as a set of code smells occurring in the modules that were modified by developers during two releases according to the data in the benchmark dataset. For ArgoUML, we prepared the oracle by first applying the source code at ver. 0.22 to the code smell detector. Then we obtained the result. Next, we used the gold set methods from the benchmark dataset for ver. 0.22 and 0.24. Finally, we intersected these two sets to obtain a list of smells that are actually related to the developer context. We applied the same process to JabRef ver. 2.0–2.6, jEdit ver. 4.2–4.3, and muCommander ver. 0.8.0–0.8.5. The dataset used for this study can be downloaded from our website[1].

Regarding the baseline of our evaluation, we used the original result from inFusion sorted by the severity of respective smells by application of a stable sort to the original result, which maintains the relative order of the results.

[1]`http://www.se.cs.titech.ac.jp/data/JSEP-smells-prioritization/`

Selection of Impact Analysis Techniques

For this study, except for RQ4.2, we used IR+Dyn for analysis because we wanted to reflect the real world situation in which the availability of information is limited. In other words, the change information and execution trace, which are inputs of IR and Dyn, respectively, are information that can be commonly found in an issue-driven development project. We excluded the technique of MSR because we wanted to confirm that our technique can function properly without prior knowledge or experience of the system, which is a requirement of MSR-based impact analyses. Regarding RQ4.2, we included the technique of MSR because we wanted to compare the results of different impact analyses. Therefore, using a greater number of techniques is preferred for investigation.

Data Analysis

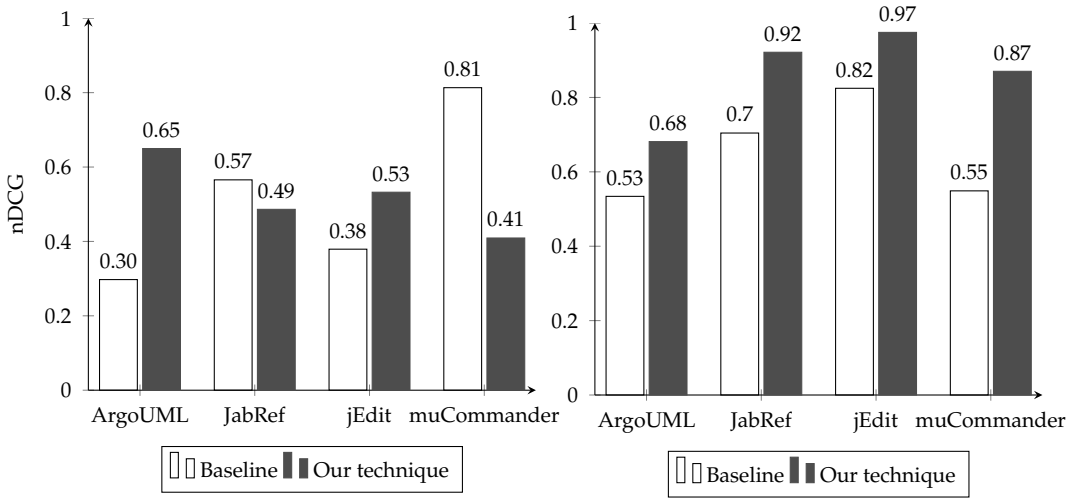
To evaluate the results, we used Normalized Discounted Cumulative Gain (nDCG) [72, 73], which is the normalization of Discounted Cumulative Gain (DCG), as a criterion. Actually, DCG is a popular measure for evaluating the quality of ranking documents with the assumption that the relevant documents appearing in the higher position of the list are more useful than those appearing in the lower position of the list. Furthermore, each relevant item can be graded by assigning a number according to the degree of relevance. DCG is calculable using the following formula:

$$DCG_p = rel_1 + \sum_{i=1}^{p} \frac{rel_i}{\log_2 i}. \tag{4.2}$$

Therein, rel_i is the graded relevance of the result at rank i; p is the length of the given ranking. Then, nDCG can be computed by normalizing DCG as

$$nDCG_p = \frac{DCG_p}{IDCG_p}, \tag{4.3}$$

where *IDCG* is the Ideal DCG, i.e., the maximum DCG value that is obtainable from the ranking result. However, because we defined the oracle as a set of code smells that might be relevant to the context of developers, the Ideal DCG, in this case, represents an approximation that might or might not be the *ideal* ranking.



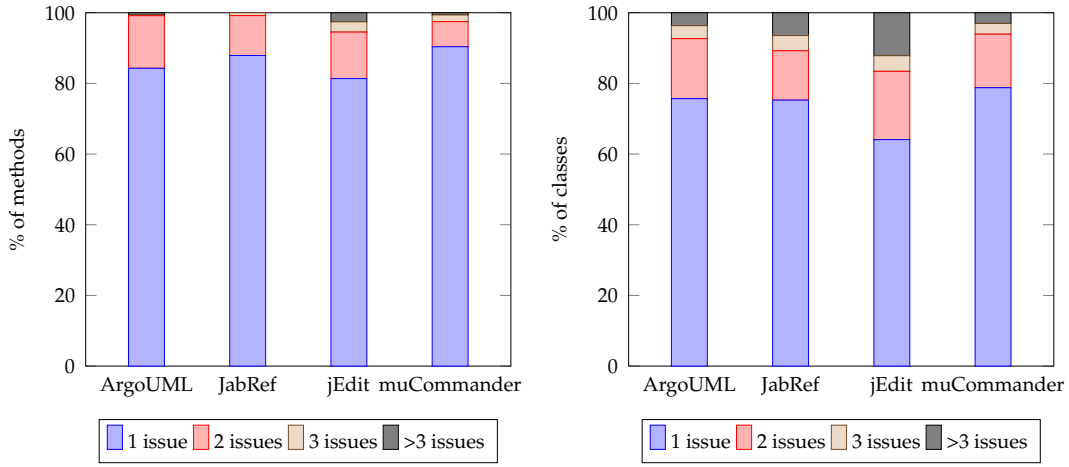
(a) Fine-grained (Method) level.

(b) Coarse-grained (Class) level.

FIGURE 4.5: Baseline nDCG values and those obtained using our approach.

The relevant documents in this study are the code smells that match the items in the oracle. The retrieved documents are the code smells in the result from the code smell detector. We defined rel_i as the number of issues in the dataset that are related to a code smell. Our assumption is that solving a code smell related to multiple issues is more useful than solving one that is unrelated or related only to one issue. Therefore, a code smell that is related to many issues is expected to have a higher degree of relevance when calculating nDCG. Because our technique involves rearranging the result from a code smell detector and assigning a higher rank to relevant code smells, the nDCG of the result from our technique is expected to be higher than the baseline of our evaluation, i.e., the original result from the code smell detector.

We calculated the nDCG of the baseline and calculated the result from our tools ordered by the CRI of each smell for all subjects.



(a) Fine-grained (Method) level. (b) Coarse-grained (Class) level.

FIGURE 4.6: Commonality between method and class levels.

4.4.2 Investigating Factors Affecting the Performance of Our Approach

RQ4.1: Which code smell granularity is more appropriate for our technique: Coarse-grained or fine-grained?

Study Design. To answer this question, we conducted two independent experiments. We applied our technique for fine-grained (method level) code smells in the first experiment, and coarse-grained (class level) code smells in the second one. We used method level impact analysis to prioritize method level code smells and class level impact analysis for prioritizing class level code smells.

Results and Discussion. Figures 4.5a and 4.5b present results of our experiments. In the fine-grained case, our technique can provide ranking qualities with the minimum nDCG 0.41 and maximum nDCG 0.65. However, in the coarse-grained case, our technique can provide ranking qualities with the minimum nDCG 0.68 and maximum nDCG 0.97. As a comparison of these two cases shows, the coarse-grained granularity code smell can yield a better ranking.

When comparing the ranking quality between the baseline and the results obtained using our technique, in the case of the class level code smell, our approach provides better ranking quality in every case. However, in the

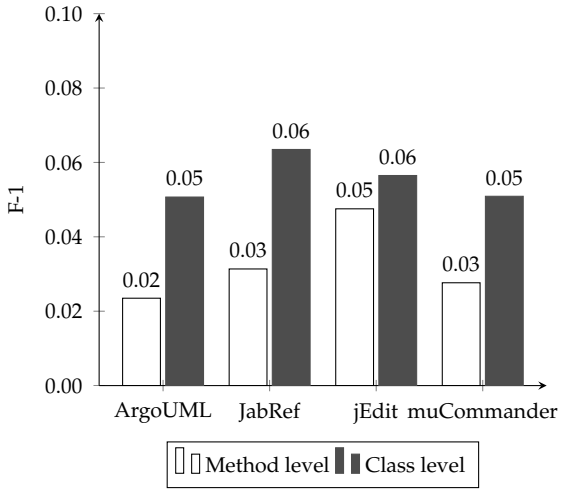


FIGURE 4.7: Accuracies of method level impact analysis and class level impact analysis.

method level code smell case, our approach generates better ranking quality than the baseline for ArgoUML and jEdit projects but fails to do so for JabRef and muCommander projects. We investigated the results, which revealed that many smells have zero *CRI*, but they are related to the items in the oracle. Therefore, our approach predicted that these smells are unrelated to the developers' context, although they actually are related. One reason for that phenomenon might be the accuracy of impact analysis. Our technique relies solely on the impact analysis result. Therefore, the accuracy of impact analysis can also affect the accuracy of our technique. The impact analysis might have failed to locate the correct module that is the target of change information. Consequently, our method incorrectly predicted that this smell is unrelated to the developer context and thereby assigned it to the lower rank of the list. The impact of the accuracy of impact analysis to our approach is discussed in the next research question. Such is not the case for class level smells because the module m from impact analysis results must be equal to or belong to *entity* of each smell s when we calculate the *CRI* value of each smell. Therefore, coarse-grained level code smells, such as class level code smells, tend to satisfy the criterion more than the fine-grained level code smells, such as method level code smells. This fact reflects that our technique is more appropriate for use with coarse-grained level code smells.

We conducted additional investigations to ascertain the reasons underlying this phenomenon. First, we analyzed the commonality of issues in the

issue tracking system. As Figs. 4.6a and 4.6b show, most of the method was modified by only one issue. More precisely, the average numbers of issues per method are 1.17, 1.13, 1.28, and 1.13, respectively, for ArgoUML, JabRef, jEdit, and muCommander. However, many classes were modified by more than one issue. More precisely, the average numbers of issues per class were 1.37, 1.46, 2.01, and 1.36, respectively, for ArgoUML, JabRef, jEdit, and muCommander. In other words, method level smells are too fine-grained regarding the commonality of issues. It is difficult to specify relevant smells for the project's context. Preferably, coarse-grained ones can be related to multiple issues. Consequently, solving code smells at the class level can be expected to contribute more to issue implementation than solving the code smell at the method level.

Then we compared the accuracy between method level impact analysis and class level impact analysis. As Fig. 4.7 shows, the F-1 scores of method level impact analyses are 115.79%, 102.46%, 18.85%, and 84.12% lower than the class level scores, respectively, for ArgoUML, JabRef, jEdit, and muCommander project. The accuracy of method level impact analysis is significantly lower than the class level accuracy, perhaps because of the nature of impact analysis technique adopted in this research. It is easier to predict a relevant coarse-grained module (class) than a relevant fine-grained module (method). Perhaps because of this reason, our technique can provide better ranking in every case of class level code smells, but it failed to do so in the case of method level code smells.

In conclusion, our technique is more appropriate with coarse-grained (class) level code smell.

RQ4.2: Does impact analysis accuracy affect the ranking quality?

Study Design. As explained earlier, work by Gethers et al. [33] shows that different impact analysis provides different accuracy. Therefore, to understand the impact of the accuracy of impact analysis to our ranking results, we used similar experimental settings to those of their work to observe differences related to accuracy among them. The techniques we used for this study include IR, IR+Dyn, IR+MSR, and IR+Dyn+MSR. Our assumption is that different techniques can be expected to influence the results of our approach.

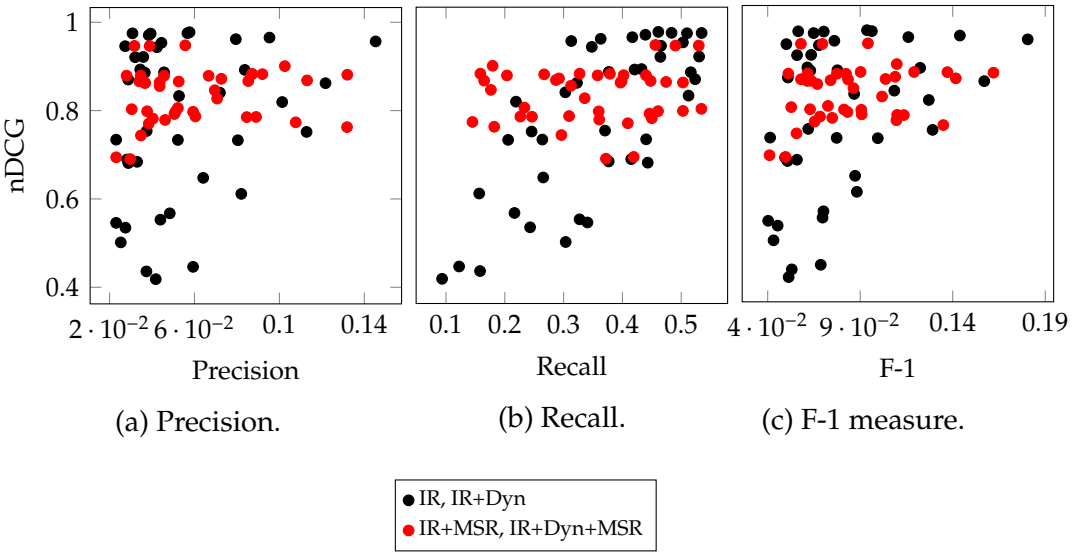
They also concluded that different cut points, i.e., the range of rankings that are going to be considered in the same technique, also contribute to the different accuracy of impact analysis. Moreover, the combination of different impact analyses might decrease the accuracy of impact analysis at certain cut points. Therefore, we also include different cut points of impact analysis (5, 10, 20, 30, and 40) into our setting.

In this study, we also divided impact analyses into two groups which are impact analyses that do and do not require prior knowledge of the system. We assume that developers who have system knowledge know at least one module that is related to the issue while the developers who have no system knowledge do not possess such information. We want to ensure that our technique can function properly in both cases. Therefore, we conducted independent experiments between MSR-related impact analyses and other impact analyses. To simulate the activity that developers select the starting point for association mining, we randomly selected the module from the gold set provided in the dataset. Definitely, the accuracy of MSR-based impact analysis depends on the randomly selected starting point. For validity, we repeated each process 100 times and used the file that is closest to the average value for calculating nDCG and accuracy.

Results and Discussion. We investigated the relation between nDCG and the accuracy of impact analysis using Spearman's correlation coefficient to evaluate the association between two variables.

Figure 4.8 shows the relation between nDCG and accuracy of each technique. The black points represent the data obtained using the IR and IR+Dyn technique, whereas the red points represent data obtained using the IR+MSR and IR+Dyn+MSR technique.

First, we consider all data points together: both black and red. By consideration of Fig. 4.8c, the relation between nDCG and the F-1 measure of each technique is apparent. The value of Spearman's correlation is approximately 0.29, indicating a weak but positive relation between the nDCG and F-1 measure. The reason underlying this weak relation might be the low values of F-1 measures. Figure 4.8a presents similar results to those of the F-1 measure. The Spearman's correlation value is approximately 0.16, indicating a very weak but positive relation between nDCG and the precision value. We also suspect another reason underlying this relation: the low precision value. However, when considering Fig. 4.8b, which represents the results of nDCG



(a) Precision. (b) Recall. (c) F-1 measure.

FIGURE 4.8: Relation between the impact analysis accuracy and the ranking quality of our technique.

value at different recall values, the relation between the two variables can be understood. The calculated Spearman's correlation value is approximately 0.48, indicating a stronger positive correlation than F-1 and precision. A tendency exists for high recall values to go with high nDCG values (and vice versa).

Then, if we consider only the MSR-related techniques (considering only red points), it is apparent that they tend to yield higher nDCG values. We suspect that this is attributable to the way we conducted the MSR approach as explained earlier. The starting points of MSR approach for association mining are randomly selected from the gold sets from each project, i.e., they are the modules that are actually going to be modified. As a result, the MSR-related techniques have a higher chance of predicting the correct modules. Consequently, they can provide higher nDCG values.

Because MSR technique requires system knowledge of the developers and manual work to specify relevant modules, we also consider the correlation of only IR and IR+Dyn technique by excluding MSR-related techniques (considering only black points). The values of Spearman's correlation are approximately 0.38, 0.19, and 0.69, respectively, for F-1, precision, and recall. It is apparent that we also obtain positive correlation between nDCG and accuracy in this case.

To sum up, when one considers all technique together, a positive correlation exists between nDCG and accuracy. However, if the MSR-related techniques are adopted, then the nDCG values are likely to be higher with the assumption that developers have system knowledge to specify the correct starting point modules. Nevertheless, if one considers only IR and IR+Dyn, which are techniques that can be fully automated using our approach, then we can also obtain positive correlations.

Therefore, we conclude that the accuracy of impact analysis tends to affect the quality of the ranking suggested by our technique. The higher the accuracy of impact analysis becomes, the better the quality of the ranking our technique can provide. Moreover, because the correlation coefficient between nDCG and recall is higher than those between nDCG and precision, and between nDCG and recall, we can infer that recall probably affects our technique the most. Therefore, our technique is more suitable for high-recall impact analysis.

In summary, the accuracy of impact analysis tends to have positive correlation with the ranking quality.

4.4.3 Studying the Meaningfulness and Usefulness of the Recommended Prioritization

RQ4.3: Does context-based smell prioritization provide more relevant results than the severity-based smell prioritization?

Study Design. We applied our technique to the dataset at the class level because it is suited to our approach, as discussed for RQ4.1. We used the combination of IR and Dyn with a cut point 40 items because, as discussed earlier, the inputs of IR and Dyn techniques can be commonly found during software maintenance. We exclude the MSR technique because we want to confirm that our technique is useful properly without system knowledge or developer experience.

Results and Discussion. As Fig. 4.5b shows, the nDCG value of the results from our technique is higher than the nDCG of the baseline. Therefore, after prioritizing the list of smells using our technique, smells that are related to

TABLE 4.4: Ranking Items for the Baseline and Our Approach.

(a) ArgoUML: Baseline.

Rank	Smell Type	Class Name	Severity	#Issues
1	Blob Class	GeneratorCSharp	8	
2	Blob Class	GeneratorJava	8	
3	God Class	FigAssociation	8	5
4	Blob Class	ParserDisplay	8	1
5	Blob Class	GeneratorPHP4	7	
6	Refused Parent Bequest	FigClassifierRole	7	3
7	Blob Class	Modeller	7	1
8	Schizophrenic Class	Import	6	
9	God Class	CoreFactoryMDRImpl	5	1
10	Refused Parent Bequest	StylePanelFigText	5	

(b) ArgoUML: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	Project	4.39	2
2	Schizophrenic Class	StylePanel	3.98	
3	God Class	ProjectBrowser	3.44	7
4	Blob Class	ProjectBrowser	3.44	7
5	God Class	ExtensionMechanism...	2.19	1
6	God Class	FigEdgeModelElement	2.05	3
7	God Class	FigNodeModelElement	1.68	4
8	Schizophrenic Class	WizStep	1.60	
9	God Class	UMLMutableGraph...	1.58	
10	God Class	UmlFactoryMDRImpl	1.58	1

(c) JabRef: Baseline.

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	BasePanel	10	4
2	God Class	JabRef	10	
3	God Class	EntryEditor	10	4
4	God Class	JabRefFrame	10	5
5	Data Class	GUIGlobals	10	
6	God Class	Util	5	7
7	God Class	EntryTableModel	5	
8	God Class	GroupsTree	5	
9	Schizophrenic Class	Option	4	
10	God Class	JabRefPreferences	4	3

(d) JabRef: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	JabRefFrame	2.94	5
2	God Class	Util	2.77	7
3	God Class	BasePanel	2.57	4
4	God Class	ImportFormatReader	1.93	
5	God Class	JabRef	1.91	
6	Blob Class	JabRef	1.91	
7	Schizophrenic Class	MainTableSelection...	1.89	2
8	God Class	EntryEditor	1.86	4
9	Blob Class	EntryEditor	1.86	4
10	God Class	BibTexDatabase	1.75	

(e) jEdit: Baseline.

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	jEdit	10	9
2	God Class	View	10	12
3	Data Class	Debug	10	
4	God Class	Buffer	10	12
5	God Class	TokenMarker	9	4
6	God Class	Gutter	9	4
7	Data Class	DirectoryEntry	8	
8	Blob Class	JEditTextArea	8	5
9	God Class	ClassGeneratorUtil	7	
10	God Class	TarEntry	7	

(f) jEdit: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	Buffer	14.50	12
2	Blob Class	Buffer	14.50	12
3	God Class	jEdit	13.38	9
4	God Class	View	11.29	12
5	Blob Class	SearchAndReplace	9.35	7
6	God Class	Gutter	9.04	4
7	Schizophrenic Class	Gutter	9.04	4
8	God Class	StatusBar	8.47	3
9	God Class	TextAreaPainter	8.18	2
10	God Class	GUIUtilities	8.12	7

(g) muCommander: Baseline.

Rank	Smell Type	Class Name	Severity	#Issues
1	God Class	JnlpTask	10	
2	God Class	StatusBar	7	
3	God Class	WindowManager	6	2
4	God Class	FileTable	6	10
5	Schizophrenic Class	CommandManager	5	1
6	God Class	FileFactory	5	3
7	God Class	HTTPFile	4	2
8	Data Class	isoPvd	4	
9	Data Class	FileCollisionChecker	4	
10	Schizophrenic Class	ActionKeymap	4	2

(h) muCommander: Our approach.

Rank	Smell Type	Class Name	CRI	#Issues
1	God Class	FolderPanel	7.21	6
2	Schizophrenic Class	LocalFile	6.34	6
3	God Class	MainFrame	4.94	3
4	God Class	WindowManager	4.28	2
5	God Class	FileTable	4.20	10
6	Schizophrenic Class	CommandManager	3.93	1
7	God Class	CommandManager	3.93	1
8	God Class	StatusBar	3.76	
9	Schizophrenic Class	StatusBar	3.76	
10	God Class	AbstractFile	3.74	2

the developers' context were on the higher rank of the list. Consequently, developers can specifically examine the top rank smells directly without specifying which smell is or is not related to their context.

Table 4.4 presents a comparison of the top ten rankings with the baseline and our approach. Each row displays the rank of the smell, type of smell, the name of the module having the smell, and the number of issues actually modified in the module. We highlighted the code smell that is actually *relevant* to the developers' context, i.e., smells in our oracle. The result of the baseline tends to be mixed with relevant and irrelevant smells, whereas our approach tends to put relevant smells at the top of the list. Moreover, the number of related issues of each smell in the baseline tends to be scattered, i.e., sometimes with a high value at the top of the list and sometimes with a low value at the top of the list. In contrast, our approach tends to put code smells with the large number of related issues at the top of the list because our assumption is that solving code smells with the large number of related issues would be more beneficial to developers, i.e., improving understandability and extensibility of the source code, than solving the code smell with a few related issues. The results can also be confirmed by calculating $nDCG_{10}$. For the baseline, we can obtain 0.22, 0.60, 0.66, and 0.35, respectively, for ArgoUML, JabRef, jEdit, and muCommander, whereas, for our approach, we can obtain 0.53, 0.85, 0.94, and 0.79 for the respective projects. The $nDCG_{10}$ obtained from the results of our approach is clearly higher than that of the baseline.

We further assessed the result by considering the 2^{nd} rank of jEdit project in the result obtained using our technique. That is the Blob Class smell of class `Buffer`. This smell was ranked 11^{th} in the baseline. However, by application of our technique, this smell became the 2^{nd} rank of the list with the highest *CRI* because this smell is related to many issues that must be resolved by the developers.

We confirmed that fact by investigating the actual changes made during release 4.2–4.3. Results showed that, out of 150 issues, 12 issues were implemented in this class, which includes this God Class smell. Therefore, if the developers realized the importance of this smell and fixed it, then it might facilitate their implementation by improving the understandability of source code for 12 issues.

In contrast, when considering the 1^{st} rank of muCommander project in the baseline which is the God Class smell of the `JnlpTask` class, it is apparent

that this smell was ranked 26th in the result from our technique. Our technique assigned this smell to the lower position of the list, specifically with zero *CRI*, because it predicted that this smell is not related in any way with any issue in the issue tracking system. We also investigated the actual change and found no issues implemented in this class. Consequently, if the developers picked the 1st smell in the original result from the code smell detector and fixed it, then it might not support their implementation for any issue at all.

This evidence indicates that a list of smells ordered by the relevance to the developers' context can support developers' implementation more than the original order such as severity.

To summarize, context-based smell prioritization provides more relevant results than severity-based prioritization.

RQ4.4: What is the effect of the combination of severity-based and context-based prioritization?

Study Design. In this study, our purpose is to study the effects of combining both severity-based and context-based approaches. Although there are many ways to achieve such combinations, e.g., using machine learning techniques, some approaches are not practical in this study because they require training data. Therefore, as a first step, we applied a simple linear combination in this study, which is expected to enable us to answer this research question. We first define the *Ranking* as a key for prioritizing code smells. The *Ranking* is the linear combination of CRI and severity. The combination is calculable as

$$Ranking = \alpha \cdot nCRI + (1 - \alpha) nSeverity, \tag{4.4}$$

where $0 \leq \alpha \leq 1$. Setting $\alpha = 0$ means that we ignore CRI and use only severity for prioritization (pure severity-based prioritization). In contrast, setting $\alpha = 1$ means that we ignore severity and use only CRI for prioritization (pure context-based prioritization). *nCRI* is the linear normalization to range 0–1 of CRI of each smell in the list. Similarly, *nSeverity* is the linear normalization to range 0–1 of severity of each smell in the list. We let α stand for a parameter for adjusting the weight of *nCRI* and severity, for example, if α is equal to 0.5, then the weight of *nCRI* and severity will be equally important.

TABLE 4.5: Ranking Items of ArgoUML between $\alpha = 1$ and $\alpha = 0.5$

(a) $\alpha = 1$.

Rank	Class Name	Ranking	#Issues	Severity
1	Project	1	2	2
2	StylePanel	0.91		3
3	ProjectBrowser	0.78	7	5
4	ProjectBrowser	0.78	7	3
5	ExtensionMechanisms..	0.50	1	2
6	FigEdgeModelElement	0.47	3	3
7	FigNodeModelElement	0.38	4	3
8	WizStep	0.37		1
9	UMLMutableGraph...	0.36		2
10	UmlFactoryMDRImpl	0.36	1	4

(b) $\alpha = 0.5$.

Rank	Class Name	Ranking	#Issues	Severity
1	ProjectBrowser	0.68	7	5
2	StylePanel	0.60		3
3	Project	0.57	2	2
4	GeneratorCSharp	0.55		8
5	GeneratorPHP4	0.55		7
6	FigAssociation	0.54	5	8
7	GeneratorJava	0.54		8
8	ProjectBrowser	0.54	7	3
9	ParserDisplay	0.5	1	8
10	FigClassifierRole	0.43	3	7

We conducted experiments to calculate nDCG for each project at different α values of 0.00–1.00 with the step of 0.01.

Results and Discussion. Figure 4.9 represents the nDCG value of the results obtained using different α values (black lines). It is apparent that they are the points at which we obtain the lowest nDCG values, as expected, because these are where we use only severity if we consider the points at which α is equal to 0. However, if one considers the points where α are equal to 1, then it is apparent that these points tend to be the points at which we can obtain high nDCG values (0.68, 0.92, 0.97, and 0.87 for ArgoUML, JabRef, jEdit, and muCommander projects). Then, if one considers the points at which α are equal to 0.5, particularly when we regard CRI and severity as equally important, we can obtain the nDCG values of 0.70, 0.81, 0.93, and 0.61 respectively for ArgoUML, JabRef, jEdit and muCommander projects. It is noteworthy that the value of nDCG is slightly lower than where α is equal to 1. However, we regard such decrements as rather small in light of the fact that we can use the severity value for prioritization.

Table 4.5 presents a comparison of the top 10 results of ArgoUML obtained when we apply our technique with $\alpha = 1$ (Context-based) and $\alpha = 0.5$ (Combination of context-based and severity-based). As discussed in earlier paragraphs, combining severity with CRI is expected to decrease the ranking quality slightly. It is apparent that the left table ($\alpha = 1$) can provide more relevant results than the right table ($\alpha = 0.5$). However, the right table tends to assign high severity code smells to the top of the list more than the left table. Specifically, the average severity of top 10 results of the left table is 2.8, whereas the right table shows average severity of 5.9.

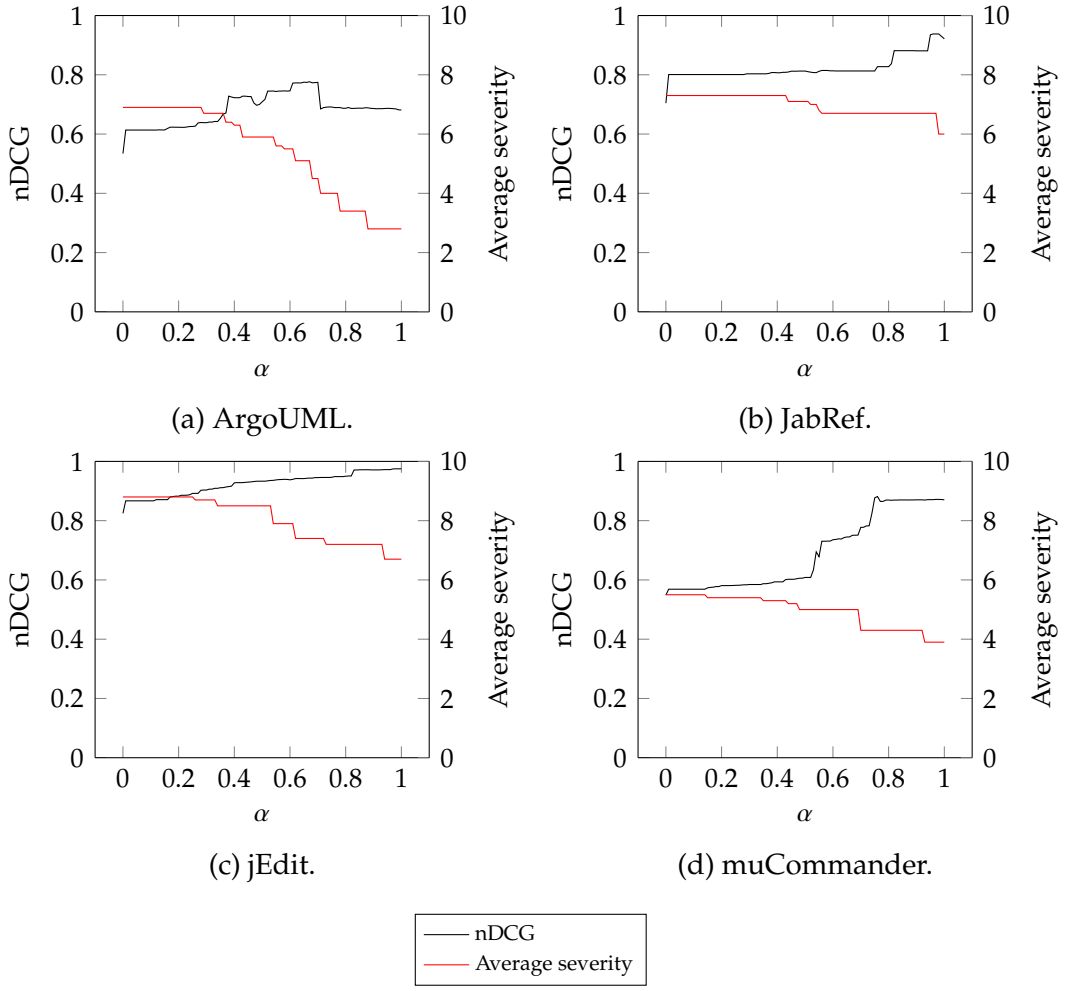


FIGURE 4.9: nDCG values and average severity of top 10 smells in the list of results obtained using different α values.

Moreover, Fig 4.9 also represents the average severity of top 10 smells in the list of the results obtained using different α values (red lines). If one considers ArgoUML project, then it is apparent that the nDCG value is approximately the same (≈ 0.69) at the point where $\alpha = 0.71$ and $\alpha = 1$. However, the average severity value of top 10 smells in the list is 4 where $\alpha = 0.71$, and 2.8 at the point where $\alpha = 1$. Therefore, in this situation, reducing the α value not only yields a higher severity value, on average; the reduction also does not affect the nDCG value. In other words, reducing α value can put more severe code smells to the top position while maintaining the context relevance of the list. The same tendency is applicable with other projects as

well.

Overall, it might be said that each combination tends to have specific characteristics based on the α value, as we expected. Low α values tend to generate results that put first code smells with high severity. This combination is useful for root canal refactoring [74], when developers aim to improve overall source code quality. In root canal refactoring, developers might not have any specific context. For that reason, particularly addressing code smells that contribute most to the decay of the system might be preferable. On the other hand, high α values are likely to produce results that put first code smells that related to the context of developers. The situation that most fits this combination is when developers perform floss refactoring [74] which is when refactoring is performed together with other changes such as bug fix or feature implementation as in prefactoring phase. In this case, developers have a specific context on the task they have to work on. For that reason, putting high priority on smells that are likely to facilitate their task is expected to yield greater benefits. We find this can be an important parameter for developers to tune the most appropriate value depending on their circumstances.

To sum up, each combination of severity-based and context-based approach tends to generate results with different characteristics. However, using α value at a certain point might be able to produce results that put first code smells with high severity while maintaining the relevance to the context.

RQ4.5: Can our approach predict the smells to be refactored?

Study Design. Bavota et al. [9] investigated the relation between the quality of a software product and related refactoring activities. They mined the history of three Java projects to verify whether developers apply refactoring operations to modules that are affected by code smells. They also provided a dataset of class level code smells that were *refactored* by developers. Therefore, we can conduct a similar study to that in Section 4.4, i.e., calculating the accuracy of ranking using nDCG. Instead of using the oracle in Section 4.4 which is a set of code smells that occur in the modules that were *modified* by developers during two releases, we used a set of code smells that occur in the modules that were *refactored* by developers during two releases. We used the number of refactorings applied to smells extracted from the dataset as rel_i when calculating nDCG because we want to see whether our technique

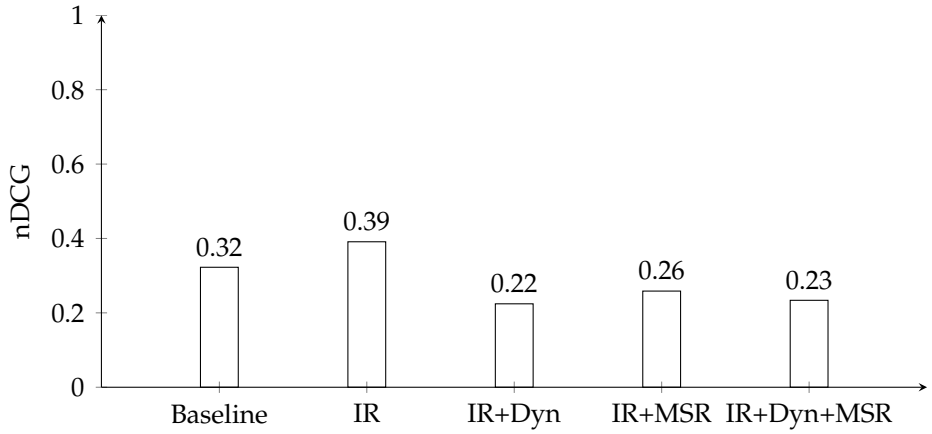


FIGURE 4.10: Accuracy of the ranking of ArgoUML using the number of refactorings applied to smells as the oracle.

can predict modules that are going to be refactored by developers. Modules that will be refactored many times are expected to have higher nDCG than those with less application of refactoring operations. Moreover, because the only project that our study and their dataset has in common is ArgoUML, we conducted the study of only ArgoUML ver. 0.22–0.24.

Results and Discussion. Figure 4.10 represents the obtained nDCG value of each impact analysis technique. We obtained the nDCG values of 0.32, 0.39, 0.22, 0.26, and 0.23, respectively, for the baseline (severity-based technique), IR, IR+Dyn, IR+MSR, and IR+MSR+Dyn (context-based techniques). In this case, we consider that all techniques produced rankings of poor quality. Therefore, we can simply conclude that our technique is unsuitable for predicting code smells that are going to be refactored by developers. The reason underlying this phenomenon is that developers in the current development style do not solve smells frequently [9]. Bavota et al. concluded that, according to their dataset, only 42% of refactoring operations are applied to modules that are affected by code smells. However, only 7% of the operations are actually code smells removed from the system. Therefore, most recommended smells on the rankings were not actually refactored, even if they *should have been* refactored. Consequently, the results of this study indicate that code smells are not the only factor that causes developers to refactor the source code. However, finding the related factors is beyond the scope of this

chapter because our intention is not to predict the smells that current software developers are solving but to recommend the smells that are expected to have sufficient impact on their development.

To conclude, our technique is inappropriate for use with predicting smells to be refactored.

4.4.4 Threats to Validity

Internal Validity

Internal validity is related to factors that might affect the outcome. In this case, the code smell detection tool that we are using, namely inFusion, detects code smells based only on object-oriented software metrics. Additionally, it might have produced some false negatives that might have an impact on our experiments. For example, if we consider modules that have metric values that are slightly below the thresholds defined using a code smell detector, then these modules are not classified as having code smells. However, if these modules are related to the context of the developers, for instance, if they are the location that developers are going to make some changes, then our technique will fail to output the results to developers even though they have poor quality because they were not in the original list. i.e., they are not *code smells* according to the detection tools.

In addition, in some cases, the detector might produce some false positives. For example, a parser class having God Class code smell is not considered problematic because its scope is generally large and because refactoring it might even reduce the comprehensibility of the class [75, 76]. To mitigate this threat, one of the authors has manually verified the top 10 results of both severity-based and context-based approaches following a false positive catalog of Fontana et al. [75, 76]. Although we did not notice any false positive in the top 10 results obtained using our approach, one false positive was found in the severity-based result of ArgoUML and JabRef project. We excluded the false positive smells and recalculated the $nDCG_{10}$. In RQ4.3, the $nDCG_{10}$ values are slightly changed from 0.22 and 0.60 to 0.29 and 0.62. In RQ4.5, the value changed from 0.22 to 0.24. We conclude that false positives have little effect on the results of this study. However, because the process of identifying false positives was performed by one of the authors who is not a main developer of the projects used for this study, we cannot ensure the

completeness of the identified result. Conducting experiments with different code smell detection techniques or applying false positive detection to the process is expected to be worthwhile. Furthermore, We rely on the dataset of Dit et al. [16] in this study. Although the dataset can be expected to have high quality because of change-based extraction, the possibility exists that errors in the dataset can affect our study results.

Construct Validity

Construct validity deals with the relation between theories and observations. The largest threat to construct validity is the oracle dataset that we used. The oracle was extracted based on the changes in version control repositories. Therefore, it might lack some important modules that were not modified by solving the given issues but which should be refactored. An example is modules that must be understood to make changes. Refactoring them contributes to the improvement of understandability. In addition, one can improve the extensibility of a module without refactoring it directly, but by refactoring another related module instead. However, it is not easy to extract such modules in an empirical manner. Version control and issue tracking repositories do not include such information in a formal way. Although the use of fine-grained interaction history of developers such as Mylyn logs [77] might be useful to confirm the context of developers more explicitly, collecting the histories of such types is another challenge [78, 79]. The details will be discussed in Chapter 4.

Another construct validity is that the *CRI* used for prioritizing code smells does not express the relevance to the *context* of the developers completely. We calculate *CRI* by relying solely on impact analysis, which takes only the change information and the source codes as inputs. Such techniques guarantee neither accuracy nor completeness. Therefore, the *CRI* that we used in this research must be improved to express the relevance to the context of the developers better.

External Validity

External validity is related to the generalization of the results. For this study, we conducted experiments on four Java open source projects available in the dataset. However, we were unable to confirm that the results will remain the same when applying our technique to different projects that might have

TABLE 4.6: Issues used for this study

Issue number	Summary
1436014	No comma added to separate keywords
1535044	Month sorting
1601651	PDF subdirectory - missing first character
1738920	Windows Position in Multi-Monitor environment
1785416	inking with exact BibKey fails

different size and language. Conducting experiments with more types of projects remains as a subject for our future work.

4.5 Controlled Experiment

As described herein, the main goal of our approach is to prioritize code smells that are related to the developers' context. We designed the approach under the assumption that code smells related to the developers' context should be solved first in the prefactoring phase because it might support the related implementation. We conducted this controlled experiment to verify whether the assumption holds, or not, and to confirm whether our approach can put first code smells that professional developers think should be solved in the prefactoring phase. Given the prefactoring phase situation, professional developers were asked to select code smells that they think should be solved. We then performed evaluations of our technique based on the list of code smells selected by developers. The next subsection explains details of our experiment.

4.5.1 Experimental Setup

Case and Subject Selection

In the previous study, we used datasets of four open source projects. Each project includes numerous issues and code smells because many developers were working on those issues. In this study, because we wanted to collect data from individual developers, using the same datasets would be impractical. Therefore, we design this study with the aim of a simulation case study

based on a real project that can minimize subjects' task while generating sufficient data for us to analyze.

We chose JabRef ver. 2.0 as our case study because its size is the smallest among four projects that we used in our previous study. We selected five issues from the dataset because we think that it is an appropriate number for a developer to analyze in a short amount of time. Our choice of issues was not random but based on several factors. First, we filtered out the issues for which the solution set of classes, i.e., those modified to solve the issue, contains no code smell because we wanted to examine modules having code smells. Then, we selected issues that modified fewer than four classes to lessen the developers' investigation time. Lastly, because we wanted the final set of issues to reflect our assumption that code smells related to many issues are more important than code smells related to few issues, we selected the final set of issues from the remaining 13 issues based on the commonality of each issue's solution set. Finally, we obtained the set of five issues shown in Table 4.6, which includes various commonality, i.e., there are two classes modified by one issue, one class modified by two issues, and one class modified by three issues.

For code smells, we filtered out code smells in the package that are irrelevant to any issue to reduce the number of code smells. We obtained total 22 smells for which the severity of each smell is 1–10. The smell types consist of Blob Class, Data Class, God Class, and Schizophrenic Class.

As for subjects, we hired 10 professional developers having experience with or currently working on Java projects. Their experience was 2–13 years. Positions of the subjects were from entry level (e.g., software engineer), senior level (e.g., senior software engineer), to management level (e.g., development group leader).

Data Collection

We first started our study by explaining an overview of JabRef application. Then, we explained the code smell concept and the definition of four code smells used for this study. The definition of prefactoring that is going to used for this study was explained afterward. The subjects were also asked to fill out a short survey about their current position and years of experience.

After completing the preparation, we explained the situation that they are assumed to be in a prefactoring phase of a particular release where they

must solve a list of issues. We then showed the subjects a list of five issues that must be solved. Each issue includes a summary and description. Because the subjects were not the actual developers of the JabRef project, asking them to find the solution of each issue independently had the potential to be time-consuming. In addition, if developers work on their projects, they are likely to be able to ascertain the solution, or the location for the solution, to the issue. Therefore, we also presented details of the solution, i.e., the *diff* of the conducted change, of each issue to fill this gap. Subjects were then asked to read and understand all five issues. Subsequently, we showed them a list of 22 code smells. Each smell includes an entity name, package name, smell type, and detailed description as to why it is regarded as a code smell, e.g., the list of methods that are the sources of the smell, provided by inFusion. Source code of the project, including the modules having code smells, were also presented to the subjects for analysis. The subjects were then asked to select code smells that should be refactored. We neither gave them any criteria nor the number of code smells they should select.

Data Analysis

Because our main goal is to evaluate the performance of our technique on prioritizing code smells that align with developers' selection, we use average precision (AP) [80], which is a popular metric for evaluating ranking documents as a criterion. The reason that we do not use nDCG as in Section 4.4 is that we regard each code smell selected by developers as equally important. Because the purpose of nDCG is to evaluate ranking documents where each document is not equally important, we found that using AP is more appropriate and straightforward for this study. AP is calculable using the following formula:

$$AP = \frac{\sum_{r \in R} P@r}{N}. \tag{4.5}$$

In that expression, R stands for the ranks of each relevant documents, N signifies the total number of relevant documents, and $P@r$ denotes the precision of the top-r retrieved documents. In this context, the relevant document is the code smell selected by developers. The retrieved document is the code smell in the result from the code smell detector. Therefore, the AP value of the

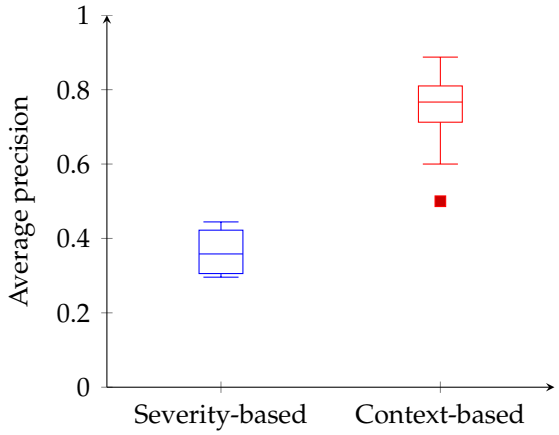


FIGURE 4.11: Average precision between severity-based and context-based approaches.

result from our technique should be high if it can put code smells that developers selected for the top of the list. Additionally, we use the mean average precision (MAP) to summarize a set of AP values.

4.5.2 RQ4.6: How does our technique prioritize code smells selected by professional developers in the prefactoring phase comparing to the severity-based approach?

Study Design

In this study, we want to evaluate our approach by comparison to the severity-based whether code smells that are put to the top of the list and by evaluating their agreement with professional developers' perspective. Therefore, we used code smells that were selected by developers as an oracle because they are what developers think should be solved during prefactoring phase. Then, we evaluated both techniques by calculating AP of each technique for each developer's answer. We made a box plot based on the obtained AP.

Results and Discussion

Figure 4.11 shows a box plot of the AP between severity-based and context-based approaches. The severity-based approach gives us a median AP of 0.36 whereas context-based approach yields 0.77. As might be readily apparent, the context-based approach outperformed the severity-based approach.

TABLE 4.7: Ranking Items for the Severity-based and Context-based Approaches.

(a) Severity-based

Rank	Smell Type	Class Name	Severity	#Devs
1	God Class	BasePanel	10	
2	God Class	JabRef	10	
3	God Class	EntryEditor	10	6
4	God Class	JabRefFrame	10	9
5	Data Class	GUIGlobals	10	
6	God Class	Util	5	9
7	God Class	EntryTableModel	5	
8	God Class	GroupsTree	5	
9	God Class	JabRefPreferences	4	9
10	God Class	EntryTable	4	

(b) Context-based

Rank	Smell Type	Class Name	CRI	#Devs
1	God Class	Util	0.46	9
2	God Class	JabRefFrame	0.28	9
3	God Class	BasePanel	0.22	
4	God Class	EntryEditor	0.21	6
5	Blob Class	EntryEditor	0.21	5
6	God Class	BibtexDatabase	0.18	
7	Data Class	GUIGlobals	0.18	
8	God Class	JabRef	0.15	
9	Blob Class	JabRef	0.15	
10	God Class	JabRefPreferences	0.13	9

We confirmed the results by application of the Wilcoxon rank sum test, which revealed that the results were statistically significant (p-value = 0.005). Therefore, our approach can generate a better ranking based on code smells that professional developers selected.

Table 4.7 presents the top 10 code smells from severity-based and context-based approaches. Each row displays the information of each smell and the number of developers that selected each smell. We highlighted the code smell that at least one developer selected during the experiment. It is noteworthy that the severity-based approach yields only four selected smells whereas the context-based yields five selected smells. In addition, the severity-based approach puts selected smells in rank as 3^{rd}, 4^{th}, 6^{th}, and 9^{th} whereas the context-based approach puts selected smells in rank as 1^{st}, 2^{nd}, 4^{th}, 5^{th}, and 10^{th}. The difference is clearer if we consider the top five items because the context-based approach includes four selected smells, whereas the severity-based approach includes only two smells. This result suggests that the context-based approach is likely to be more practical for use in the prefactoring phase.

In addition, if we calculate AP using the oracle that we used in the previous study, i.e., a set of code smells occurring in the modules that were modified by developers, we can obtain a value of 0.42 from severity-based and 0.81 from the context-based approach. By comparing these values to the MAP obtained from the oracle used for this study, which are 0.36 and 0.74 for severity-based and context-based respectively, we can infer that the results in this study support the results from our empirical studies for which we use a set of code smells occurring in the modified modules as an oracle.

In conclusion, results show that our context-based approach can prioritize code smells selected by professional developers better than the severity-based approach can.

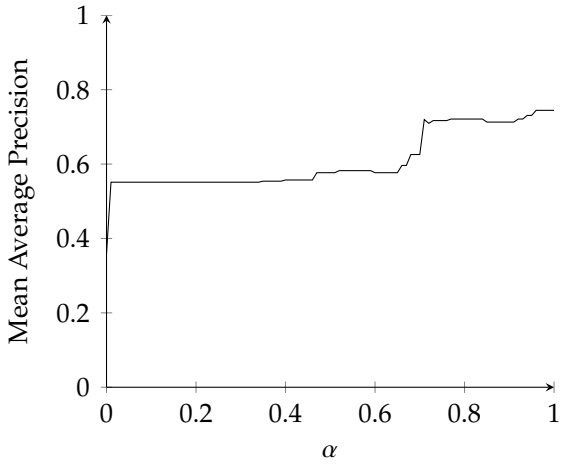


FIGURE 4.12: Mean Average Precision of results obtained using different α values.

4.5.3 RQ4.7: What is the most appropriate proportion of linear combination of severity-based and context-based prioritization for the prefactoring phase?

Study Design

In RQ4.4, we studied the effect of the combination of the severity-based and context-based approaches. We concluded that high α value in Equation 4.4 is appropriate for the situation in which developers who have a specific context perform refactoring to support the implementation such as prefactoring phase. Consequently, regarding this research question, we conducted experiments using different α values. Then, we calculate MAP, which is the mean of AP of each developers' answer at different α values. This is expected to yield α that can generate the best MAP which is the most appropriate α value for prefactoring phase.

Results and Discussion

Figure 4.12 represents the MAPs of respective α values. In this case, it is apparent that the value of MAP tends to increase where α value increases, perhaps because of the fact that developers selected code smells to be refactored by relevance to the context, not by severity. Therefore, in this study,

we conclude that using α value close to 1 is likely to yield best ranking results for prefactoring. The result here agrees with those of our previous study described in Section 4.4 showing that the prefactoring phase is more appropriate with high α value.

To sump up, α value close to 1 is most appropriate for the prefactoring phase.

4.5.4 Threats to Validity

Internal Validity

A salient concern might be the fact that the subjects are not main developers of JabRef project. Consequently, they might not have a deep understanding of the system. However, because we selected only issues that require simple modification (fewer than four locations) and because we provide them the solution of each issue as a guideline for understanding changes, we believe that we could mitigate this threat. Nevertheless, providing the subjects with the diffs of the solutions might have an impact on the results because the subjects might select code smells by biasedly map with the class name in each solution. Conducting an experiment with the core developers of the targeted project without providing the solution of the issues might be able to mitigate this threat.

Another threat to internal validity might be the final set of issues to be used for this study. When selecting them, aside from filtering out by other factors, we strove to include various commonality of the issues, i.e., some issues modified the same classes and some issues modified different classes. The reason is, as discussed, to reflect our assumption that smells related to many issues are more important than smells related to few issues. Therefore, selecting only those issues that modify the same class would not be able to verify the assumption and would be less beneficial.

External Validity

For this study, we conducted experiments on a reduced size of issues and code smells which might be a threat. However, asking developers to perform tasks on the whole set of data maybe impractical because it requires a considerable amount of time and effort. In addition, although the results of this study are similar to the results of the previous study for which we used

the full set of data, we cannot ensure if the results are going to be applicable to other sets of issues. Such investigation is beyond the scope of this study and remains as a subject of our future work. The number of developers in this study is 10, which might not be representative. However, the subjects in our study are from a different background within a range of 2 to 13 years of experience. The positions of the subjects are also varied from junior, senior, and management level. Nonetheless, conducting the study of a larger scale might be beneficial.

4.6 Related Work

Since the concept of code smell has been introduced, many techniques have been proposed to facilitate the process of code smell detection. Nonetheless, because code smell detectors tend to generate huge numbers of smells, many techniques have also been proposed to reduce the number of code smell detection results, e.g., prioritization and filtration. This section summarizes work related to code smell detection, prioritization, and filtration. Moreover, we conclude with presentation of some work in the literature that considers the developer context for supporting developers.

4.6.1 Code Smell Detection

Research related to code smell detection has devoted the most attention to using metric values of source code for detection of code smells such as detection strategies reported by Marinescu [38], DECOR by Moha et al. [23], jDeodorant by Tsantalis et al. [39], an approach by Munro et al. [40], and an approach by Lanza and Marinescu [3]. In addition, many attempts have been made at applying other techniques to enhance metric-based analysis. For example, Khomh et al. [81] proposed a Bayesian approach that can work with incomplete data and which allows analysts to adjust the results with their knowledge. Oliveto et al. [82] proposed a technique based on numerical analysis to overcome the code smell detector limitations that either have too strict rules or which require expensive tuning by an expert. Sahin et al. [83] proposed code smell detection as a bilevel problem that allows the prediction of new code smells that differ from those of an example. Moreover, recent research such as that by Fontana et al. [26, 84] has underscored the use of machine learning for detecting code smells.

Although most works specifically examine metric-based detection, some attempts have been conducted to use information of different types to detect code smells. Palomba et al. [25] proposed TACO as a technique that uses textual analysis to detect code smells. Their technique applies information retrieval to textual elements such as source code identifiers and comments and calculates the probability that a code component is affected by a code smell. Additionally, they proposed HIST, which uses change history information mined from version control systems to detect code smells [24]. Specifically, they exploit association mining to analyze co-changes and to generate modules that are likely to be affected by code smells.

As we discussed earlier, because we designed our approach as a framework, it is useful with most code smell detectors. This benefit is expected to be more useful than strict reliance on a detection strategy.

4.6.2 Code Smell Prioritization

Arcoverde et al. [45] proposed a technique to prioritize code anomalies, or code smells, based on their potential to the software architecture degradation. They used four heuristics: change density, error density, anomaly density, and architecture role. They concluded that their technique is mostly useful in scenarios of several kinds: architectural problems in a system involving classes that changed together, architectural problems related to communicating across different classes, changes that are not predominantly perfective, some architecture roles that are affected by many code smells, and the architecture roles of the system are well defined and have distinct architectural relevance. In contrast, our technique emphasizes support of developers during the prefactoring phase. We devote attention only to smelly modules within the focus of developers because solving smells on the modules that developers are not going to touch would not support their implementation. Moreover, we limit our input to the issue tracking system to reflect real-world situations in which the availability of information is limited.

Fontana et al. [41] introduced the Code Smell Intensity index as a criterion to prioritize code smells. The Code Smell Intensity index is computed by the distribution of software metrics and is intended to quantify the importance of each code smell instance. They presented the index as a numeric value of 1–10 and defined five intensity levels: Very Low, Low, Mean, High, and Very High. The approach devotes attention specifically to the most severe smell

instances. Moreover, it is limited to code smells of six types: God Class, Data Class, Brain Method, Shotgun Surgery, Dispersed Coupling, and Message Chain. In contrast, our approach specifically examines more context-related smell instances. Furthermore, ours is not limited to specific smells. The novel point of our approach compared to those others is that our approach is applicable to smells of many kinds. We prioritize every smell in the detection result based on the relevance to the developers' context.

Vidal et al. [46] proposed a technique to prioritize code smells based on three criteria: historical information of component modification, the relevance of the type of code smell from the developer's perspective, and modifiability scenarios of the system. The technique prioritizes code smells based on the context of developers using modifiability scenarios. However, such information requires experience and knowledge of the system, as well as manual work specifying scenarios and related source code components. In contrast, our technique can process this step automatically. Their technique also requires some information related to the change history of the system. Such information would mostly be available in the late stages of software development, where our technique is useful at any stage of software development. Consideration of other factors such as code smell type preferences of developers remains a subject of our future work.

Codabux and Williams [85] presented a framework for prioritizing technical debt using predictive analysis. They first consider classes that are defect-prone and change-prone as a technical debt item. They then calculate the *technical debt proneness* of each class using a prediction model that was constructed using a Bayesian Approach. The items are categorized into low, medium, and high groups using the prediction model. Finally, with the decision from project managers or developers, their technique generates technical debt items having the probability of being the most critical. As described earlier, their technique considers classes that are defect-prone and change-prone as a technical debt item. However, our technique specifically examines every code smell generated from detection tools. Moreover, their work requires a manual decision of the developers to generate output, whereas our technique emphasizes context-related results and process this step manually.

4.6.3 Code Smell Filtration

Fontana et al. [42] proposed a technique to reduce the number of code smell detection results by the application of strong and weak filters, which is calculated by consideration of the application domain of the system. A strong filter is intended to remove false positives from code smells detection results such as code smells in a test class. A weak filter identifies code smells that might not be a shortcoming of the system or which do not affect software quality, such as the Shotgun Surgery smell in getter and setter methods so that developers can solve them when they have time. However, the method limits the technique to code smells of five types: God Class, Data Class, Shotgun Surgery, Dispersed Coupling, and Message Chains, whereas our technique has no such limitation. Moreover, the aim of their technique is to improve the accuracy of code smell detection, whereas our technique intends to provide developers with relevant code smells.

Ratiu et al. [59] used historical information to increase the accuracy of code smell detection. They measured the stability, i.e., how often the class changed (at least one method was added or removed) and persistence, i.e., how long the class has been affected by code smells, of classes. Then they used the results to filter out the entities that might not adversely affect the original results detected using a single-version strategy. They also identified most dangerous smells using additional analyzed historical information. However, their technique is limited to God Class and Data Class code smells. Similarly to the works described earlier, the approach specifically examines improvement of the accuracy of smell detection by filtering out smells that are unlikely to cause problems with the system, whereas our technique emphasizes the context of developers.

4.6.4 Context-based Approach

Hayashi et al. [86] proposed a technique to suggest refactoring operations using a sequence of program modification. Their work uses the editing operation by which a developer performs during the program modification activities, e.g., copy paste, to estimate the developer's context. The technique specifically examines suggesting refactoring operations only on the modules that the developer is currently working on without devoting attention to other modules, even though they are affected by code smells.

Liu et al. [87] presented a monitor-based instant refactoring framework to suggest refactoring opportunities to a developer. The technique will run in the background during the code changing process of the developer. The technique will then invoke the code smell detector and warn developers to solve the code smells if such changes are likely to introduce code smells. In other words, this technique is designed to detect code smells only on the modules on which developers are currently working.

Morales et al. [88] proposed ReCon, an approach that uses a developer's context for automatic refactoring of code smells. They defined the refactoring strategies for code smells of four types: Lazy Class, Long Parameter List, Spaghetti Code, and Speculative Generality. Then they used the *task context* to limit the process scope. The task context is the source code entities that developers used when working on maintenance task. Such information can be extracted from monitoring tools log such as Mylyn. They used metaheuristics techniques, which are Simulated Annealing, Genetic Algorithm, and Variable Neighborhood Search, to generate the sequence of refactoring operations that can remove code smells.

Even though the resources for context estimation differ, the techniques described earlier, and our technique specifically examines the *current context* of the developer. Nevertheless, although our work specifically examines supporting the *prefactoring* phase, when developers refactor source code to prepare for implementation, their work can be regarded as supporting the *postfactoring* phase because the technique works during the source code editing process used by developers.

4.7 Conclusion

As described herein, we proposed a technique for prioritizing code smell detection results by consideration of the developers' current context. We estimated the context of the developers by application of impact analysis to the list of issues in issue tracking system and used the results to calculate the CRI of each code smell. The results obtained using our technique are a list of smells, prioritized based on their relevance to the developer context. Smells that are more relevant to the developer context are ranked higher on the list. Therefore, our approach can assist developers in prioritizing code smells for

the prefactoring phase. Our technique is useful for planning how to prefactor the source code before implementing sets of issues in an issue-tracking system.

We conducted empirical studies of four open source projects related to the use of developers' context and smell prioritization to study the characteristics of our technique and the factors that may affect the result's quality. First, results show that coarse-grained code smells can provide a better ranking than fine-grained code smells for several reasons. Second, the accuracy of impact analysis tends to affect the ranking quality. Third, results show that our context-based smell prioritization technique can provide more relevant results than the severity-based smell prioritization. Fourth, each combination of severity-based and context-based approach is appropriate for different situations. Finally, results show that, because the developers in the current of software development style do not solve smells frequently, our approach is inappropriate for predicting smells that are going to be refactored.

Additionally, we conducted a controlled experiment and showed that our technique can put first code smells that are agreed with developers' choices. We also concluded that the combination approach with high α (combination ratio) value is more practical in prefactoring phase.

Chapter 5

Using Context-Based Code Smells Prioritization to Support Referred Context

Summary

Because numerous code smells are revealed by code smell detectors, many attempts have been undertaken to mitigate related problems by prioritizing and filtering code smells. We earlier proposed a technique to prioritize code smells by leveraging the context of the developers, i.e., the modules that the developers plan to implement. Our empirical studies revealed that the results of code smells prioritized using our technique are useful to support developers' implementation on the modules they intend to change. Nonetheless, in software change processes, developers often navigate through many modules and refer to them before making actual changes. Such modules are important when considering the developers' context. Therefore, it is essential to ascertain whether our technique can also support developers on modules to which they are going to refer to make changes. We conducted an empirical study of an open source project adopting tools for recording developers' interaction history. Our results demonstrate that the code smells prioritized using our approach can also be used to support developers for modules to which developers are going to refer, irrespective of the need for modification.

This chapter is based on our paper "Revisiting Context-Based Code Smells Prioritization: On Supporting Referred Context" [58] published at the Ninth International Workshop on Managing Technical Debt (MTD'17),

5.1 Introduction

Code smells are often interpreted as indicators of problems or design flaws in source code [1]. They can be seen as factors that cause technical debts. For instance, a class having a God Class code smell is a class that tends to be packed with functionality from other classes and which tends to control data of other classes. As a result, the class is likely to be overly large and complex. This kind of design flaw can have a distinctly negative effect on software maintenance. It affects many perspectives of source code quality such as understandability and extendibility. Many studies have been conducted to investigate code smell effects [19, 20].

One important problem of current code smell detection is that the number of outputs is overwhelming. Consequently, developers do not use static analysis tools such as code smell detectors because of numerous detected warnings [14]. To resolve this shortcoming, numerous attempts have been made to reduce the number of code smells using different factors by filtering and prioritizing the code smells [41, 45, 46, 85].

Nevertheless, with the limited time available for solving code smells or improving the quality of the source code in real world situations, it is preferable to solve code smells in the modules related to the context of the developers, as reported in the literature. One characteristic that Murphy-Hill and Black presented in their "Seven Habits of a Highly Effective Smell Detector" paper [89] is *Context-Sensitivity*, which is to suggest code smells that are related to the current context of developers first. Yamashita et al. [17] also stated that developers in their study need code smell detectors that support context-sensitivity. Furthermore, Bavota et al. [9] recommended in their work that perspectives of developers need to be considered when recommending refactoring opportunities.

In Chapter 4, we proposed a technique to prioritize code smells based on the developers' context [44]. The definition of *context* described herein is similar to *task context* defined by Kersten and Murphy as "the information–a graph of elements and relationships of program artifacts–that a programmer needs to know to complete that task" [10]. We estimated the context of developers by application of an impact analysis technique to textual information, e.g., summary and description of issues in an issue tracking system, to predict the modules that are likely to be locations for making changes to resolve issues. Then, we prioritized the list of code smells based on results obtained

using an impact analysis technique. We evaluated our technique using a set of code smells occurring in the modules that were actually modified by developers as the oracle obtainable from source code repositories such as Git or Subversion. The results confirmed that our technique can be used to support modules that are going to be modified by developers.

However, to make changes, it is common that developers start the process by navigating through multiple places and referring to them before they actually modify the source code. We designated the modules to which developers are going to refer as *referred modules* and modules to which developers must make actual modifications as *modified modules*. When considering the developers' context to prioritize code smells, it is rational to estimate the modified modules as the context because those modules are the locations at which developers are going to make modifications directly. Nonetheless, the referred modules are also important to be used for estimation of the context because those modules are the modules where developers are going to read and comprehend although they require no modification. Consequently, it is important to investigate whether the context-based prioritization technique is useful to support both situations when considering the referred modules and the modified modules as the context of developers. However, in Chapter 4, we only studied and confirmed that the technique might be useful for situations involving modified modules. As a consequence, one important point remains in our context-based prioritization technique: whether our technique can support not only the modified modules but also the referred modules.

To bridge the gap as described earlier, we conducted an empirical study of the use of our technique to support the referred modules. Such information is obtainable using interaction-recording tools such as Mylyn [77]. Our result confirmed that our technique can be used to support the referred modules as well. In other words, if developers solve code smells according to the results of our technique, then it can help them improve several aspects of source code quality in the related modules. Consequently, it might reduce the cost for their implementation.

The main contribution of this work is a demonstration that the result of context-based code smells prioritization is useful not only to support developers during modifications but also during navigation and reference to source code for resolving issues in an issue tracking system.

The structure of this chapter is organized as follows. Section 5.2 presents

a brief introduction to source code interaction history. We then report our empirical study in Section 5.3, discuss threats to validity in Section 5.4, and conclude this chapter in Section 5.5.

5.2 Mylyn's Interaction History

Software development teams have been widely adopting version control systems such as Git or Subversion to keep track of source code changes. However, one shortcoming of the version control system is that it records only code components that have been modified at the commit time, while in fact, developers also refer to many code components that they might not have actually modified. To overcome such a lack of information, many interaction history recording tools such as Mylyn [77] have been proposed. Mylyn records developers' activities on the tasks that developers work on, for instance when they select text in the editor. Such interaction events are recorded in XML file format and are uploaded to the issue tracking system of each issue on which the developers worked. The attributes of each element of interaction event that we are considering include the following[1].

- **Kind**: type of interaction
- **StructureKind**: type of artifact interacted upon by developers
- **StructureHandle**: the artifact interacted upon by developers

In Mylyn, when developers select some text in the editor, the interaction event in which the value of *Kind* is *Edit* is going to be recorded. However, such information is insufficient to ascertain whether developers were referring only to the source code or were also modifying the source code. Therefore, to classify the referred and the modified modules, we used this information together with the information from a version control system. We regard the modules in the version control system as modified modules and regard the modules in the interaction history that are not in the version control system as the referred modules.

[1] `https://wiki.eclipse.org/Mylyn/Integrator_Reference`

5.3 Empirical Study

5.3.1 Motivation

As described herein, our study in Chapter 4 used the oracle created solely based on changes in the version control system. Such information only contains modules that were changed at the commit time. However, it is very likely that there are also many other modules to which developers are going to refer even though they need not make any modification. Developers might need to understand those modules to be able to make the actual modification. Therefore, we want to confirm through this study whether the list of code smells that is prioritized by our technique can be useful to support not only the modules that developers are going to modify but also the modules to which developers are going to refer.

5.3.2 Study Design

Experimental Implementation

As in Chapter 4, we implemented an automated tool for conducting the experiment. Our tool was designed to connect with other tools such as TraceLab [16, 67] for carrying out impact analysis and inFusion [30] for detecting code smells used for this study.

To generate the gold sets or the modified modules based on the version control system, we used tools proposed by Dit et al. [16]. Their tools generate gold sets by comparing two versions of each file based on Eclipse's Abstract Syntax Tree (AST). The result of the tools is a list of modules that were modified for each commit.

Data Collection

For this study, we used the Mylyn Task project as our subject because Mylyn projects contains a large amount of Mylyn interaction history information [90]. We mined over 700 commits between ver. 3.07–3.21 from its version control system[1]. Then, because the developers of Mylyn projects are recommended to use Eclipse's plugin to commit changes to Git repository and

[1] `https://github.com/eclipse/mylyn.tasks`

because the commits include the URL of the bug that the developers are solving, we were able to create links between commits and issues by searching for the pattern of corresponding bug's URL (such as `https://bugs.eclipse.org/bugs/show_bug.cgi?id=500712`) for each commit message. We created 516 links that included 334 unique issues. Subsequently, we crawled Mylyn's issue tracking system according to the URL that we obtained from commit messages. Because not every developer uploaded the interaction history to the issue tracking system, we had to filter out issues that did not contain interaction information. The reason behind this is that the interaction history was necessary for our study to analyze the modules that developers did not modify but referred to. In all, we obtained 95 issues that satisfy our requirements.

We defined oracles of two types: modification-based and reference-based. The modification-based oracle is a set of code smells that appear in the modules that were modified by developers, while the reference-based oracle is a set of code smells that appear in the modules that were referred to by developers. When we generated the oracle, out of 141 detected code smells, besides code smells that are not related to the context, we obtained 37 code smells in modification-based oracles and 61 code smells in reference-based oracles. In addition, the code smells that are only in the reference-based oracle are 27. The code smells that are only in the modification-based oracle are 3.

Data Analysis

For the assessment of this analysis, we used Normalized Discounted Cumulative Gain (nDCG) [73], which is a popular metric for evaluating the quality of ranking documents. nDCG assigns a higher value to relevant documents appearing in the higher position on the list than the relevant documents appearing in the lower position of the list. Additionally, we can assign the degree of relevance to each relevant item. Therefore, in this study, we can use nDCG to reflect the quality of our technique, which is used to assign code smells that affect many issues at the top of the list. We calculated the nDCG value for both the modification-based oracle and the reference-based oracle using Equations 4.2 and 4.3 presented in Chapter 4. For the modification-based oracle, rel_i is the number of issues that modify a module having code

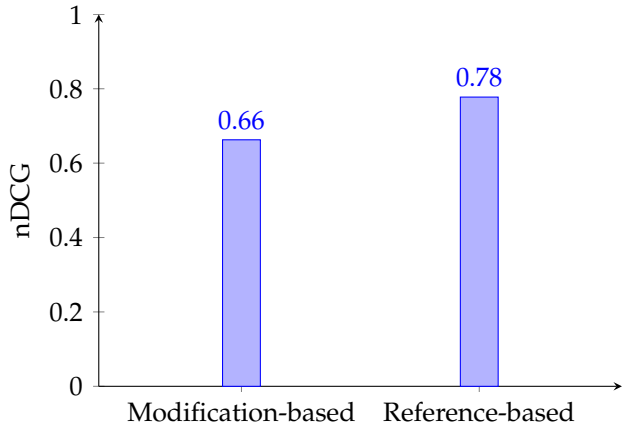


FIGURE 5.1: Comparison of the nDCG value between results of modification-based oracle and reference-based oracle.

smells. For the reference-based oracle, rel_i is the number of issues that referred to a module having code smells.

5.3.3 Results and Discussion

Figure 5.1 presents the results of our study. We obtained an nDCG value of 0.66 in the case of a modification-based oracle and 0.78 in the case of a reference-based oracle, which indicates that our technique not only prioritizes code smells that have an effect on the modules that developers are going to modify but also prioritizes code smells that have an effect on the modules to which developers are going to refer. Therefore, if developers solve code smells according to the list provided by our technique, then it is going to support them, e.g., make the code easier to understand, not only when they are modifying source code, but also when they are referring to the source code.

Table 5.1 presents the top 10 results of the code smells after being prioritized with our technique together with the number of issues that refer to and modify the module having the code smells. For instance, our technique predicted that the God Class code smell of `TasksUiInternal` class is related to many issues that developers must solve with *CRI* of 7.89. Our investigation of the actual interaction history of developers during the analyzed period revealed that for seven issues, developers referred to this class without actually modifying it. Subsequently, we investigated the change history from the version control system, which revealed that, for four issues, developers

TABLE 5.1: Top 10 Prioritized Code Smell Results

Rank	Smell Type	Class Name	CRI	#RI[a]	#MI[b]
1	God Class	`TasksUiInternal`	7.89	7	4
2	God Class	`TasksUiPlugin`	5.51	9	3
3	God Class	`TaskListIndex`	5.48	3	1
4	God Class	`AbstractTaskEditorPage`	5.36	3	2
5	God Class	`TaskDataManager`	5.29	4	2
6	God Class	`TracRepositoryConnector`	5.26	1	1
7	God Class	`AttachmentUtil`	5.24	4	1
8	God Class	`SynchronizeTasksJob`	5.17	3	0
9	Data Class	`TaskData`	4.97	3	0
10	God Class	`BugzillaRepositoryConnector`	4.69	0	2

[a]Number of referring issues
[b]Number of modifying issues

had to modify this class to resolve the issues. Therefore, if developers solve the code smell of this class before starting the implementation process, then several perspectives of this class, such as understandability and extendibility, can be improved. Consequently, the implementation costs of 11 issues related to this class can be reduced.

In addition, if we consider the God Class smell in `SynchronizeTasksJob` class in the table, then it is apparent that no issue exists in the analyzed period that modified this class. Consequently, solving the code smell in this class does not support any code modification process of the developers. However, the analysis result shows that there are three issues to which developers referred to this class, even though they did not modify it. Therefore, although solving this code smell might be unable to support developers for the code modification, it can support developers when they refer to this code, i.e., improve the source code comprehensibility.

Furthermore, if we analyze the number of oracles as described earlier, then it is apparent that the code smells that are only in the reference-based oracle are 27. Such a number is high compared with other numbers such as code smells that are only in the modification-based oracle, which are only 3. Therefore, we conclude that the referred modules have a strong effect on developers' implementation process and that they should be considered when considering the developers' context. In addition, because our technique can predict numerous code smells that are in reference-based oracle (e.g., most

items in top 10 ranks are predicted correctly), we can reach the conclusion that our context-based code smells prioritization can suggest not only smells in a modified context but also those in a referred context.

5.4 Threats to Validity

As this study is a preliminary study, there are some threats to validity that need to be addressed. First, we conducted the study on a small scale with only one project. Second, while other interaction recording tools might be able to used, we used only Mylyn for this study, and it might have caused false negatives. Finally, we used only one metric to assess this study. Nonetheless, we planned to justify these threats by conducting a larger scale study with different kinds of datasets and tools in the future. Additionaly, performing different evaluation schemes such as different kinds of metrics or with professional developers remains as a subject for our future work.

5.5 Conclusion

As described herein, we ascertain whether our context-based code smells prioritization can be useful to support a situation in which developers refer to source code modules. We conducted an empirical study with an oracle based on a version control system that represents the modules which developers modified and an oracle based on the interaction history, which represents the modules to which developers referred to resolve the issues. Our results confirmed that the context-based code smells prioritization can support situations in which developers modify and refer to source code.

Chapter 6

An Exploratory Study on Decaying Modules for Proactive Refactoring

Summary

Source code quality is often measured using code smell, which is an indicator of design flaw or problem in the source code. Code smells can be detected using tools such as static analyzer that detects code smells based on source code metrics. Further, developers perform refactoring activities based on the result of such detection tools to improve source code quality. However, such approach can be considered as *reactive refactoring*, i.e., developers react to code smells after they occur. This means that developers first suffer the effects of low quality source code (e.g., low readability and understandability) before they start solving code smells. In this study, we focus on *proactive refactoring*, i.e., refactoring source code before it becomes smelly. This approach would allow developers to maintain source code quality without having to suffer the impact of code smells.

To support the proactive refactoring process, we propose a technique to detect *decaying modules*, which are non-smelly modules that are about to become smelly. We present empirical studies on open source projects with the aim of studying the characteristics of decaying modules. Additionally, to facilitate developers in the refactoring planning process, we perform a study on using a machine learning technique to predict decaying modules and report a factor that contributes most to the performance of the model under consideration.

This chapter is based on our paper "Toward Proactive Refactoring: An Exploratory Study on Decaying Modules" [91] published at the Third International Workshop on Refactoring (IWoR'19), Copyright © 2019 IEEE.

6.1 Introduction

Code smells were introduced as an indicator of a design flaw or problem in the source code [1]. Their definitions were presented in a descriptive language; therefore, several studies have interpreted them in a formal manner. For example, Lanza and Marinescu use source code metrics to form conditions and combine each condition with logical operations to detect code smells [3]. Several studies have found that code smells are related to different aspects of software development such as maintainability [19, 51, 54]. Therefore, it is advisable to remove code smells by a refactoring operation that can improve the quality of the source code and avoid undesirable consequences.

However, such approach can be considered as *reactive refactoring*. We term it reactive because developers basically react to code smells after they occur in the system. An advantage of this approach is that it allows developers to focus on the most problematic part of the source code (smelly code) rather than handling every part of the source code, which may be impractical in real life. However, an important disadvantage of such approach is that, by the time developers are warned of the code smells by the tools, they have already suffered the bad effect of the code smells, e.g., low readability and understandability. In other words, developers cannot prevent code smells from occurring in the system.

To deal with the problem, in this study, we shift our focus to *proactive refactoring*, which is the action of refactoring source code before it becomes smelly. Similar to the manner in which code smells are considered as candidates for performing reactive refactoring, we propose the idea of *decaying modules* as candidates for performing proactive refactoring. A decaying module is a non-smelly module that is about to become smelly. We measure the quality index of a module by calculating the *module decay index* (*MDI*) that indicates the closeness of a module to becoming smelly. MDI is used to measure how bad a non-smelly module is, whereas *severity* [43] and *smell intensity index* [42] are used to measure how bad a smelly module is. The idea of decaying module can be mainly used in two ways. First, it can be used to warn developers of the modules that are getting close to becoming smelly so that developers can take preventive measures. Second, it can be used to indicate overall quality of the entire system so that developers can view the overall status of a project, and not just the smelly modules. This would allow developers to develop more proactive strategies for controlling software quality.

In other words, developers can focus on preventing code modules from becoming smelly rather than waiting until they become smelly and then resolve the code smells.

The main contributions of this study are as follows.

1. We propose the concept of decaying modules by observing MDI.
2. We present empirical studies regarding characteristics of decaying modules.
3. We report an experiment on using a machine learning approach to predict decaying modules and show that developers' context can significantly improve the performance of the prediction model.

The remainder of this chapter is organized as follows. Section 6.2 presents the definition of decaying modules. Section 6.3 presents our empirical studies on decaying modules. Section 6.4 presents an experiment on the prediction approach. Section 6.5 discusses the threats to validity of this study. Section 6.6 presents the related works. Section 6.7 concludes this chapter.

6.2 Decaying Module

6.2.1 Motivation

To describe the motivation behind the concept of decaying module, we use a dental metaphor because it has been used to explain the idea of software quality, e.g., floss and root-canal refactoring [74]. We draw a parallelism between the process of removing code smells and tooth decay treatment, i.e., developers handle code smells after they occur; this is similar to the act of patients taking care of their tooth decays after they happen. On the contrary, the process of preventing code smells is comparable to the process of brushing one's teeth every day, i.e., developers prevent code smells from occurring by refactoring modules that are not yet smelly; this is similar to the act of brushing one's teeth to remove small food particles. However, it is impractical to refactor every module in the system; therefore, we define the concept of *decaying module* to represent a module without code smell but with progressively worsening quality. If a tooth without tooth decay is building up plaque, then the plaque becomes the primary cause of the tooth decay in

the future. Considering the aforementioned example, the concept of decaying module can be used to support proactive refactoring, i.e., developers can refactor decaying modules to prevent them from becoming smelly.

The main objective of decaying modules is to identify modules with a risk of becoming smelly in the future. This may be related with an empirical study by Tufano et al., which reported that modules that are likely to be affected by code smells are characterized by specific metrics' trends [92]. Accordingly, we suspect that observing the distance between the metric values and the thresholds indicating that the module will be considered as smelly would enable us to generate a set of modules that are likely to be affected by code smells.

6.2.2 Definition

We define a decaying module as a module that is getting closer to becoming smelly during a certain period. One way to detect code smells is to use a metric-based strategy. Such a strategy detects code smells by considering whether particular metrics exceed their corresponding thresholds. In this case, we can use a formula to detect a decaying module. A decaying module would be a module whose metric values have not exceeded their thresholds. Therefore, we refer to code smell detection strategies that use multiple symptoms (or conditions) to detect code smells. Each symptom is determined by whether a source code metric, e.g., lines of code (LOC) exceeds a specific threshold. Logical operations are then used to combine all the symptoms and identify code smells. For example, the strategy to detect God class defined by Lanza and Marinescu [3] can be reformulated as follows.

$$\text{God class} = (V_{\text{ATFD}} \geq T_{\text{ATFD}}) \wedge (V_{\text{WMC}} \geq T_{\text{WMC}}) \wedge (V_{\text{TCC}} \leq T_{\text{TCC}}) \qquad (6.1)$$

where V_{ATFD}, V_{WMC}, and V_{TCC} are the values of access to foreign data (ATFD), weighted method count (WMC), and tight capsule cohesion (TCC), respectively. Similarly, T_{ATFD}, T_{WMC}, and T_{TCC} are the thresholds of ATFD, WMC, and TCC, respectively.

This strategy includes three symptoms, where each symptom is determined by a corresponding metric. For example, the metric ATFD measures how many foreign attributes are used by a class. Higher the ATFD, more

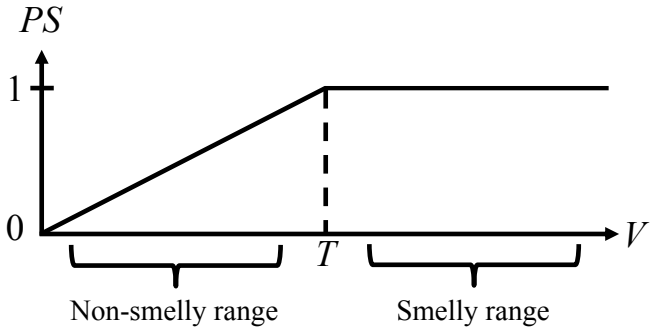


FIGURE 6.1: Domain and range of percentage of symptom (PS).

likely is a class to be a God class. Therefore, the operation ≥ is used to determine if the value of ATFD exceeds the threshold; if the former is true, the symptom holds. On the contrary, the metric TCC represents the degree of cohesiveness of a class. Lower TCC value indicates a less cohesive class; this means that the class is more likely to be a God class. Here, the operation ≤ is used to determine if the value of TCC is less than the threshold; if the former is true, the symptom holds. All the symptoms are then conjunctively combined using *and* operations to determine the God class code smell.

Then, we define the following metrics to measure the closeness of a module to becoming smelly.

Percentage of Symptom

First, we define a *percentage of symptom* (*PS*) to measure how close the current value of a specific metric is to its threshold. PS is measured by metric m of module x, which is defined as:

$$PS_m(x) = \begin{cases} \min\{1, V_m(x)/T_m\}, & \text{if comparator is } \geq \\ \min\{1, T_m/V_m(x)\}, & \text{if comparator is } \leq, \end{cases} \tag{6.2}$$

where V_m is the value of metric m and T_m is the threshold value of metric m. The min function sets the maximum value of PS to one.

For example, PS measured by ATFD of module x can be defined as

$$PS_{\text{ATFD}}(x) = \min\{1, V_{\text{ATFD}}(x)/T_{\text{ATFD}}\} \tag{6.3}$$

because the operator that determines its symptom is $\geq$. On the contrary, PS measured by the TCC of module x can be defined as

$$PS_{\text{TCC}}(x) = \min\{1, T_{\text{TCC}}/V_{\text{TCC}}(x)\} \tag{6.4}$$

because the operator that determines its symptom is $\leq$.

Figure 6.1 shows the domain and range of PS. Higher the PS, closer it is to complete the condition of the symptom. PS of one indicates that the symptom is completed.

Module Decay Index

Next, we define *MDI* as an indicator of how close a module is to becoming a code smell. A MDI is defined for each smell; MDI of smell s can be calculated by averaging the PS of each symptom measured by metric m where $M_s = \{\ldots, m, \ldots\}$ as:

$$MDI_s(x) = \frac{1}{|M|} \sum_{m \in M_s} PS_m(x). \tag{6.5}$$

For example, the MDI of God Class of module x can be defined as

$$MDI_{\text{God class}}(x) = \frac{1}{3}\{PS_{\text{ATFD}}(x) + PS_{\text{WMC}}(x) + PS_{\text{TCC}}(x)\}. \tag{6.6}$$

Higher the MDI, closer is the module to becoming a code smell. MDI of one indicates that the module is a code smell.

MDI is also comparable to severity defined by Marinescu [43]. Severity is a metric ranged $[1, 10]$, and it is used to measure how bad a smelly module is. MDI, however, ranges from $[0, 1)$ with the purpose of measuring how bad a non-smelly module is. In other words, MDI can also be considered as severity of non-smelly modules.

Decaying module

In general, a decaying module can be defined as a module whose MDI has increased over a period of time. In this chapter, we refer to a module as decaying module when its current MDI has increased from the previous release.

As a first step in defining decaying modules, we use God class as a subject owing to its simple detection strategy and the fact that it is a code smell that is often studied in this research area [11]. However, this approach is also applicable to other types of code smells that are determined by using logical operations to combine all symptoms, where each symptom holds if particular metrics exceed their threshold such as data class or brain class defined by Lanza and Marinescu [3].

6.3 Decaying Module: Empirical Study

6.3.1 Motivation

This empirical study aims to conduct analyses on characteristics of a decaying module from different perspectives.

First, we empirically investigate the number of decaying modules that occur between releases. We expect that such information can be used as a first step to understand the characteristics of decaying modules. The number of modules is different depending on the size of the project; therefore, analyzing absolute numbers may be not useful. We analyze the ratio of decaying modules to the number of modules that are modified by developers in each release. This is because module modification is the main activity during software development, and it is relatively easy to understand as a comparator.

Second, we study the characteristics of decaying modules, especially, their future characteristics. In this study, we consider two characteristics of decaying modules, i.e., the decaying modules that will be modified and will decay in the future. If the number of decaying modules that will decay in the future is higher than that of non-decaying modules, it may be a sign that decaying modules are important problems that are worth handling.

6.3.2 Research Questions

The empirical study was conducted with the following research questions.

RQ6.1: How many decaying modules are present in each release compared to the modified modules?

RQ6.2: What are the future characteristics of the decaying modules compared to the non-decaying ones?

TABLE 6.1: Dataset Information

Project	Period	# Releases
Accumulo	2012/03/27 – 2018/07/16	30
Ambari	2013/02/04 – 2018/08/22	33
Derby	2005/08/01 – 2018/05/05	28
Hive	2010/10/27 – 2018/05/18	33

Details of each research question are explained later.

6.3.3 Experimental Setup

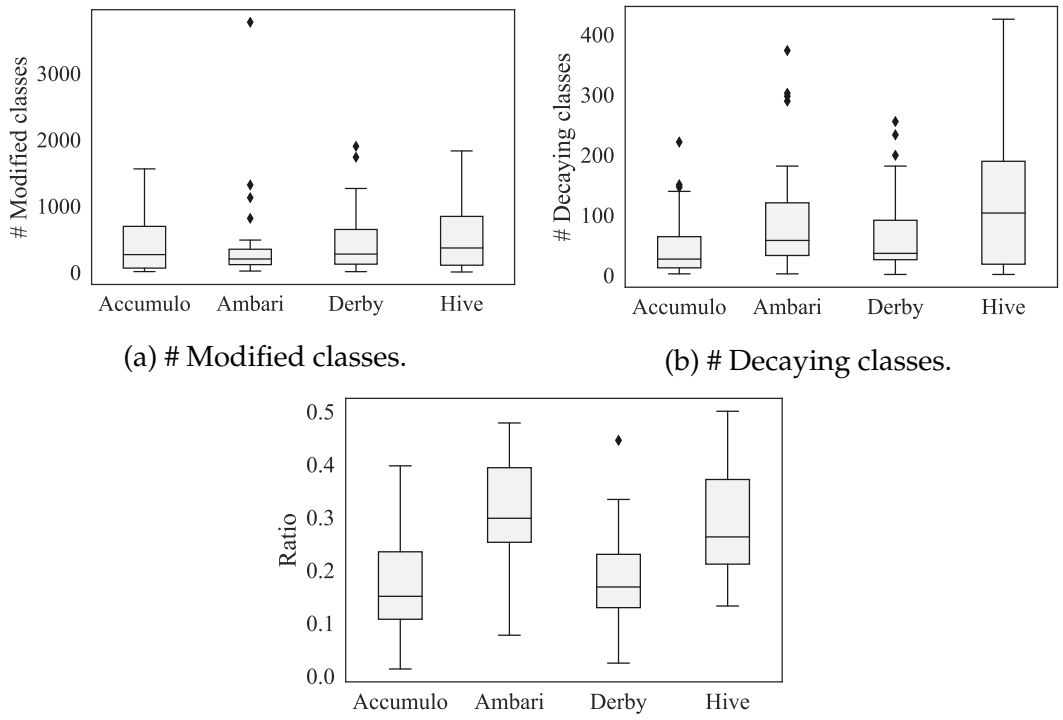
Experimental Implementation

In this study, as a first step in studying decaying modules, we limit the target of the code smell to be the God class. As discussed earlier, God class is considered one of the most common code smells studied in this research area. To detect the decaying module, we first used inFusion ver. 1.9.0 as a static analysis tool to calculate the metrics of each module. Then, we calculated the *PS* of each metric and *MDI* of each module. Finally, we classify those modules as decaying modules whose *MDI*s have increased from the earlier release.

Data Collection

In this study, four open source projects: Accumulo[1], Ambari[2], Derby[3], and Hive[4] were our subjects. They were selected from a list of active open source projects of The Apache Software Foundation, which is commonly used as a subject for open source software study. The dataset information can be found in Table 6.1. Accumulo is a low-latency, large table data storage and retrieval system. Ambari is a software that allows system administrators to manage and monitor a Hadoop cluster. Derby is a relational database management system written entirely in Java. Lastly, Hive is a data warehouse software built on top of Hadoop.

[1] `https://accumulo.apache.org/`
[2] `https://ambari.apache.org/`
[3] `https://db.apache.org/derby/`
[4] `https://hive.apache.org/`



(a) # Modified classes.

(b) # Decaying classes.

(c) Ratio.

FIGURE 6.2: The number of decaying modules, modified modules, and the ratio between them.

6.3.4 RQ6.1: How many decaying modules are present in each release compared to the modified modules?

Study Design

To answer this research question, we counted the number of modules that were modified and the number of decaying modules between each pair of releases. The modules that were modified between each pair of releases were generated using the `git log` command. Then, we ran our tool that was explained previously to detect the decaying modules between each pair of releases. Finally, we calculated the ratio between the number of modified and decaying modules.

TABLE 6.2: Averages of the Number of Decaying and Modified Modules

Project	# Modified classes	# Decaying classes	Ratio
Accumulo	427.17	50.68	0.12
Ambari	394.34	98.47	0.25
Derby	450.78	72.48	0.16
Hive	627.94	132.17	0.21

Results and Discussion

Figures 6.2a, 6.2b, and 6.2c represent the number of modified classes, the number of decaying classes, and the ratio of them respectively. For the number of modified classes, we can observe similar distributions for all projects except for Ambari, which tends to have the lowest number of modified classes among all projects. For the number of decaying classes, we can see that Hive has the largest distribution among four projects. Finally, for the ratio, we can observe the similar distributions between Accumulo and Derby, and between Ambari and Hive.

To simplify the results, we summarize the averages of each value in Table 6.2. The second and third columns show the average numbers of the modified and decaying modules, respectively. The last column shows the ratio of the number of modified and decaying classes. For example, in each release in the Accumulo project, 427.17 classes were modified and 50.68 were decayed. This yields a ratio of 0.12, i.e., for every 100 modified classes, 12 classes were decayed. The ratios are varied for different projects: 0.12 for Accumulo, 0.25 for Ambari, 0.16 for Derby, and 0.21 for Hive. The average ratio in this study is approximately 0.19. In other words, compared to modified modules, 19% will become decaying modules. This result suggests that the decaying modules are not rare problems and may be worth considering as essential problems that the developers should handle by considering that almost 20% of the modified modules will have lower code quality.

In conclusion, approximately 19% of the number of modified modules were decaying modules in each release on average.

6.3.5 RQ6.2: What are the future characteristics of decaying modules compared to non-decaying ones?

Study Design

Similar to RQ6.1, we counted the decaying modules with the following two characteristics: 1.) the decaying modules that will be modified in later releases, and 2.) the decaying modules that will again get decayed in later releases. Then, we computed the averages of all releases and calculated the ratio of each. For comparison, similar steps were applied to the modules that were modified but did not become decaying modules (i.e., non-decaying modules). We excluded the releases without decaying modules, e.g., minor releases that have only few modifications, because we cannot compare the ratios of decaying and non-decaying modules.

To confirm whether the results are statistically significant, we conducted statistical tests with the following null hypotheses:

$\mathbf{H_{01}}$: The ratios of decaying modules that will be **modified** in later releases are *not higher* than the ones of non-decaying modules.

$\mathbf{H_{02}}$: The ratios of decaying modules that will get **decayed** in later releases are *not higher* than the ones of non-decaying modules.

Accordingly, the following alternative hypotheses corresponding to each null hypothesis can be defined as follows:

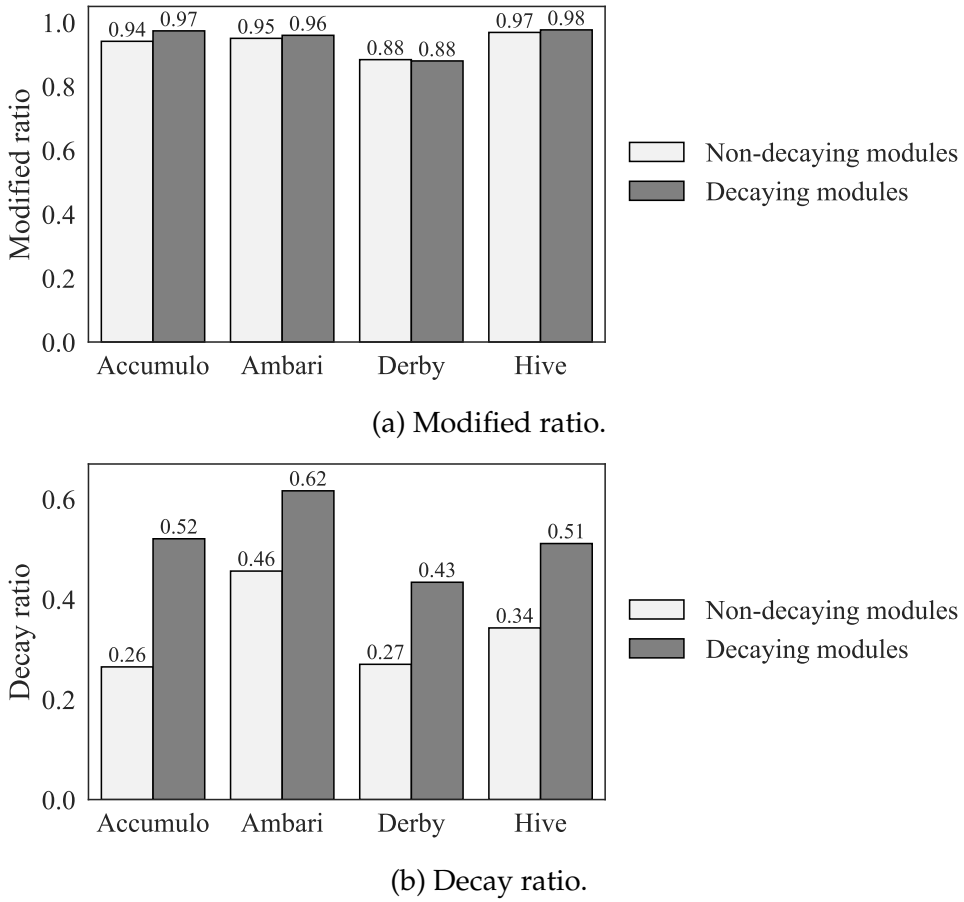
$\mathbf{H_{a1}}$: The ratios of decaying modules that will be **modified** in later releases are *higher* than those of non-decaying modules.

$\mathbf{H_{a2}}$: The ratios of decaying modules that will get **decayed** in later releases are *higher* than those of non-decaying modules.

Then, we used the Wilcoxon signed-rank test, which is a non-parametric statistical hypothesis test, to determine any difference between the decaying and non-decaying modules.

Results and Discussion

Figures 6.3a and 6.3b illustrate the results of our study. The bars represent the values of non-decaying and decaying modules. The values in Fig. 6.3a represent the ratio of modules that will be modified in later releases. We



(a) Modified ratio.

(b) Decay ratio.

FIGURE 6.3: Comparison of averages of the number of decaying and non-decaying classes that will be modified and decay.

can observe only small, if any, differences in the projects. For example, the biggest difference is in the values of the Accumulo project indicating that approximately 94% and 97% of the non-decaying and decaying modules, respectively were modified in later releases. On the contrary, both the values of the Derby project are approximately 88% with slightly higher number of decaying-modules modified in the later release. Table 6.3 shows the result of Wilcoxon signed-rank test. The cells with p values less than 0.05 are highlighted in gray. It can be seen that, while the p values of Accumulo and Hive are less than 0.05, the p values of Ambari and Derby are higher than 0.05. This means that for Ambari and Derby we fail to reject the null hypotheses ($\alpha = 0.05$) that the ratios of decaying modules that will be modified in later

TABLE 6.3: Results of Wilcoxon Signed-Rank Test

Project	H_{01}	H_{02}	H_{03}
Accumulo	0.038	<0.001	<0.001
Ambari	0.130	<0.001	<0.001
Derby	0.571	<0.001	<0.001
Hive	0.046	<0.001	<0.001

releases are not higher than the ones of non-decaying modules. For Accumulo and Hive, although we can reject the null hypotheses, the differences are very small.

Next, Fig. 6.3b illustrates the ratios of modules that decayed in later releases. For example, in the Ambari project, 46% of the non-decaying modules decayed in later releases. However, the number for decaying modules is as high as 62%. The ratios are clearly higher than those of their non-decaying counterparts. In other words, decaying modules are more likely to get decayed again in later releases. Additionally, it can be observed that more than half of the decaying modules in each project decayed again in later releases. The results of the Wilcoxon signed-rank test listed in Table 6.3 indicate that the results are statistically significant ($\alpha = 0.05$). In such cases, we can reject the null hypotheses that the ratios of decaying modules that will get decayed in later releases are not higher than the ones of non-decaying modules.

With these pieces of evidence, we can conclude that, although we could not observe the difference of the ratios of decaying and non-decaying modules that will be modified in later releases, the ratios of decaying modules that will get decayed in later releases is higher than those of non-decaying modules. In other words, we can conclude that decaying modules are the modules that have got closer and will likely get more close to becoming smelly modules. Therefore, we argue that they are significant problems that should be handled before they affect source code quality.

In conclusion, while no difference of the ratios of decaying and non-decaying modules that will be modified in later release was detected, decaying modules are more likely to get decayed in later releases.

6.4 Decaying Class: Prediction

6.4.1 Motivation

As discussed in the previous section, decaying modules are important problems that are worth considering. However, a decaying module only represents past and present information, i.e., whether a module has decayed from the last release. As shown in the previous section, although decaying modules are more likely to get decayed again in the future, they also have a chance of not getting decayed in the next release, i.e., their quality may be improved. Therefore, knowing that a decaying module will still be a decaying module in the next release (its quality will get even lower), can support developers to determine whether it should be refactored. Therefore, in this study, we consider the use of a machine learning approach to predict modules that will become decaying modules in the next release. Such approach can be implemented by considering the characteristics of each module (e.g., code quality metrics) as predictor variables and whether a module will get decayed in the next release (*True* or *False*) as a response variable. This technique is widely used for defect prediction, where the characteristics of each module are used to predict whether the module is defective [93]. Another example that is similar to this study is the work by Pantiuchina et al. [94] that proposes to predict code smells. Their work uses source code quality to predict whether a module is likely to be affected by code smells in the future. Therefore, in this study, our aim is to investigate whether their approach is also applicable to predicting decaying modules. If such an approach can successfully apply to predicting decaying modules, we can use its result to support developers when selecting targets for refactoring.

A scenario that is suitable to this case is the prefactoring phase, wherein developers refactor the source code to facilitate future implementation [15]. In this scenario, developers can use the prediction result of the modules that will get decayed in the next release to plan their refactoring strategy. In prefactoring phase, developers usually have an idea of the changes that they plan to make, i.e., they know the modules that they are going to change. Such information is obtained from the concept location and the impact analysis phase, wherein developers identify the code component that needs to be modified to satisfy change requirement [15]. We define this as *developers' context* [47]; it is similar to the *task context* defined by Kersten and Murphy as

"the information–a graph of elements and relationships of program artifacts–that a programmer needs to know to complete that task" [10]. In this case, we suspect that developers' context may contribute to improving the performance of prediction models. The underlying reason is that the modules that will be modified are more likely to get decayed than the modules that will not be affected by any changes.

6.4.2 Research Question

In this section, we presented an empirical study with the following research questions.

RQ6.3: Can we use an existing technique to predict decaying modules?

RQ6.4: Does developers' context improve prediction performance?

Details of each research question will be discussed in the later subsections.

6.4.3 Experimental Setup

Baseline

We set a baseline model inspired by the work by Pantiuchina et al. that was proposed to predict modules that will become smelly [94]. Three sets of variables are used as predictor variables: current code quality, historical code quality trend, and recent code quality trend. The response variable is whether the module will get decayed in the next release, i.e., after implementing changes that are planned for the release.

Variables Construction

We calculated three types of predictor variables: current code quality, historical code quality trend, and recent code quality trend. Historical code quality trend represents the trend of each module's quality from its creation until the current release, whereas recent code quality trend represents the trend of each module's quality from the earlier release until the current release. These two types of variables can be used to supplement each other in case when the quality of a module decreased in the past but increased during recent activities. For the variable: current code quality, we use inFusion ver 1.9.0 as a

TABLE 6.4: Metrics Used in This Study

Label	Metric Name
ALD	Access to Local Data
AMW	Average Method Weight
ATFD	Access to Foreign Data
BOVR	Base-class Overriding Ratio
BUR	Base-class Usage Ratio
CBO	Coupling Between Objects
CPFD	Capsules Providing Foreign Data
CRIX	Criticality Index
CW	Class Weight
DIT	Depth of Inheritance Tree
MDI	Module Decay Index
HIT	Height of Inheritance Tree
ICDO	Incoming Coupling Dispersion for an Operation
LCC	Loose Capsule Cohesion
LCOM	Lack of Cohesion of Methods
LDA	Locality of Data Accesses
LOC	Lines of Code
NAS	Number of Added Services
NOA	Number of Attributes
NOACCM	Number of Accessor Methods
NOAM	Number of Abstract Methods
NOCHLD	Number of Children
NOM	Number of Methods
NOPRTA	Number of Protected Attributes
NOPRTM	Number of Protected Methods
NOPUBA	Number of Public Attributes
NOPUBM	Number of Public Methods
NOVRM	Number of Overriding Methods
OCDO	Outgoing Coupling Dispersion for an Operation
PNAS	Percentage of Newly Added Services
RFC	Response for a Class
SPIDX	Specialization Index
SUM_LOC	Sum of Lines of Code
TCC	Tight Capsule Cohesion
WOC	Weighted Operation Count

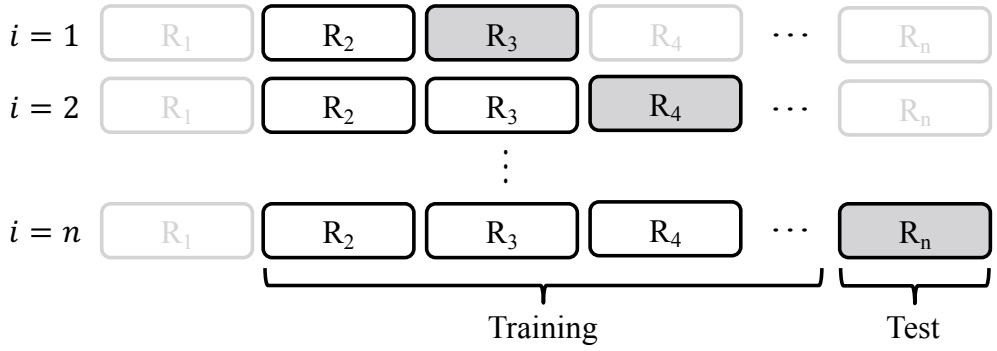


FIGURE 6.4: Training and test data separation.

static analysis tool to calculate 34 metrics in addition to the proposed *MDI*. The metrics used in this study are shown in Table 6.4. For the variable: historical code quality trend, we compute the regression slope line fitting the value of each metric from the first to the current release. Finally, for the variable: recent code quality trend, we compute the regression slope line fitting the value of each metric from the earlier to the current release. As a result, we have a total of 105 (35 + 35 + 35) predictor variables.

Consequently, we used the technique proposed in the previous section to detect decaying modules and record as response variables.

Data Separation

We separated data into training and test sets as shown in Fig. 6.4. Initial releases of the source code were skipped because there is insufficient information to calculate the slope of each metric. In the first iteration (i = 1), we trained the model on release 1, and performed a prediction test on release 2. Then, in the next iteration (i = 2), we trained the model on release 1–2, and performed a prediction test on release 3. The iterations were repeated for the whole dataset. This strategy simulates the situation where developers use all of the available data to train the model. Such data may be fewer at the beginning of a project but would increase along with the development process.

Data Preparation

Next, we performed correlation-based feature selection technique to minimize collinearity among the predictor variables. For each pair of variables

having Spearman ρ higher than 0.8, we removed one of the variables. The technique was repeatedly conducted until there was no pair of variables that met the criteria. Totally, we removed 16, 19, 15, and 19 variables for the projects: Accumulo, Ambari, Derby, and Hive, respectively.

Additionally, the dataset that we use in this study can be considered as imbalanced, i.e., the number of decaying modules is only a small proportion of all the modules. To avoid the problem of imbalanced data affecting the performance of the prediction models, we applied a sub-sample technique to the dataset. For each training set, we randomly selected non-decaying modules equal to the number of decaying modules to balance the classes of response variables.

Model Construction

Finally, we constructed prediction models using the random forest method [95] of the scikit-learn library [96] and calculated performance of the prediction models by applying them to the data in the test sets. We kept default values for the model parameters. Optimizing and analyzing such parameters remains our future work.

Data Analysis

In this study, we use area under the curve (AUC) of the receiver operating characteristic (ROC) plot, which is commonly used for evaluating and comparing the performance of the machine learning model. AUC ranges from 0 to 1. Higher AUC indicates better performance of the prediction model. Its value above 0.5 indicates that the model performs better than random guessing.

6.4.4 RQ6.3: Can we use an existing approach to predict decaying modules?

Study Design

We use the baseline model explained in the previous subsection to measure prediction performance. As aforementioned, AUC value above 0.5 indicates

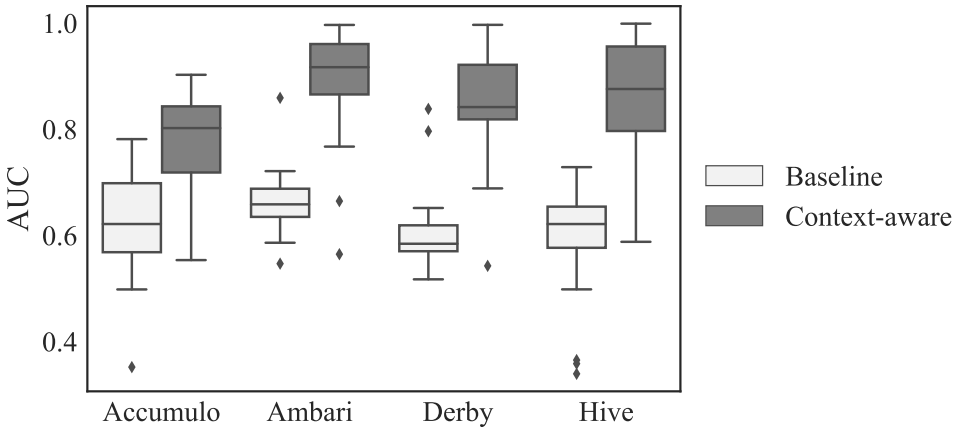


FIGURE 6.5: AUC values of baseline and context-aware models.

that prediction performs better than random guessing. Therefore, if the median of AUC measured by the baseline is greater than 0.5, we infer that an existing approach can be used to predict decaying modules.

Results and Discussion

Figure 6.5 shows the performance of the baseline model of each project. The median values are 0.62, 0.66, 0.58, and 0.62 for Accumulo, Ambari, Derby, and Hive, respectively. Median values of all the projects are greater than 0.5; therefore, we can conclude that the existing approach may be suitable for predicting decaying modules as well.

To conclude, an existing approach may be applicable to predict decaying modules.

6.4.5 RQ6.4: Does developers' context improve prediction performance?

Study Design

We compare the performance of two models: baseline and context-aware. The baseline model uses the variables described previously as predictor variables, whereas the context-aware model uses the developers' context as an extra predictor variable. As aforementioned, we regard developers' context as the modules that the developers intend to modify for the next release. We use `git log` command to obtain a list of modules that are modified between

two releases. Then, we mark a variable as *True* if the module is modified, and *False* otherwise. It is noteworthy that this way of representing developers' context may not be practical as it assumes perfect knowledge of developers. However, our main goal of this preliminary study is to examine the potential of using developers' context to improve the prediction model. In other words, we want to know the upper bound performance of using developers' context. Then, we plan to conduct a study with more practical settings, e.g., using automated approaches such as impact analysis to estimate developers' context, in the future.

To confirm if the results are statistically significant, we conduct the Wilcoxon signed-rank test with the following null hypothesis:

$\mathbf{H_{03}}$: Developers' context does *not improve* the performance of prediction model.

Therefore, an alternative hypothesis can be defined as:

$\mathbf{H_{a3}}$: Developers' context *improves* the performance of prediction model.

Results and Discussion

Figure 6.5 shows the result of our experiment. The box plot shows the values of the performance of the baseline and context-aware models. As reported in the RQ 6.3, median values of the AUC values for the baseline model are 0.62, 0.66, 0.58, and 0.62 for Accumulo, Ambari, Derby, and Hive, respectively. For the context-aware model, the median values are 0.80, 0.92, 0.84, and 0.88 for Accumulo, Ambari, Derby, and Hive, respectively. It can be seen that the context-aware model performs better for every project.

The results of the Wilcoxon signed-rank test are shown in Table 6.3. It can be seen that the results are statistically significant ($\alpha = 0.05$). Therefore, we can conclude that developers' context can help improve decaying module prediction performance significantly.

In conclusion, developers' context can improve the performance of the decaying module prediction model.

6.5 Threats to Validity

In this study, we use four open source projects as our subjects. Therefore, the results of this study may not generalize to other types of projects. Additionally, we conducted the experiment on the prediction model using only random forest method without performing any parameter optimization. Therefore, the result may differ in different models and different parameter settings. It is noteworthy that the primary goal of this study is not to find the highest performance of the prediction model but to show that existing techniques from a different research area can also be applied to this problem and that developers' context can improve the performance significantly. Moreover, replicating this study on a larger scale may be beneficial.

6.6 Related Work

Murphy-Hill and Black used the terms *floss refactoring* and *root-canal refactoring* to refer to different refactoring tactics [74]. They use frequency and how developers mix refactoring with other kinds of program changes to categorize the two types of refactoring. Floss refactoring refers to frequent refactoring that is mixed with other types of program changes, whereas root-canal refactoring refers to infrequent refactoring that may not be mixed with other types of changes. On the contrary, in this study, we use the timing and purpose of refactoring to categorize the two types of refactoring. We term it reactive refactoring if the operation is applied after a code smell occurs in the source code with the purpose of removing the code smell and proactive refactoring if the operation is applied before a code smell occurs in the source code with the purpose of preventing a code smell. Therefore, floss refactoring and root-canal refactoring can be considered to be both reactive and proactive depending on when and why the developers perform the refactoring.

One of the work aligned with the idea of proactive refactoring is *just-in-time refactoring* proposed by Pantiuchina et al. [94]. They proposed an approach to predict code components that will be affected by God and complex classes' code smells within a specific time. The approach allows developers to prevent code smells by refactoring source code right before they are introduced to the system. On the contrary, the idea of decaying module and module decay index (MDI) proposed in this study can not only be used to

prevent the introduction of code smells but also to represent the status of the source code quality of non-smelly modules using the context of code smells.

The MDI proposed in this study can also be compared to *severity* [43] and *smell intensity index* [42] which were proposed to measure how bad a code smell is. Such metrics can be used to prioritize code smells, i.e., more severe code smells should be refactored first if developers want to improve the overall quality of systems. MDI can be used in a similar manner; however, for non-smelly modules. In other words, it can be used to measure the quality of non-smelly modules so that developers can notice the quality of the whole system, and not only smelly modules. Such information may allow developers to develop new strategies of refactoring by looking at the whole system and preventing modules from decaying rather than only reactively remove code smells from the system.

The similar term, *code decay*, is defined by Eick et al. in their work as "Code is decayed if it is more difficult to change than it should be" [97]. They use effort, interval, and quality as the keys to identify code decay. However, in this study, our definition is closely related to code smells, i.e., decaying modules are modules that are getting closer to becoming smelly. *code decay indices (CDIs)* are also defined to quantify symptoms as risk factors. Although CDIs are mostly computable directly from a version control system, MDI defined in this study is computed from each module metrics value and the threshold of the symptoms of the code smell.

Additionally, in addition to representing the location that should be refactored, code smells are also used to describe characteristics of source code. For instance, Takahashi et al. successfully used code smell severity to improve the accuracy of bug localization [98]. Nevertheless, it is obvious that such a technique cannot be used on non-smelly modules as there is no such metric to indicate the quality of non-smelly module. Thus, in addition to being used for supporting proactive refactoring, we expect MDI to be used to describe characteristics of source code that can be applied to other research fields as well.

6.7 Conclusion

In this study, we propose the idea of decaying module that can be used to support proactive refactoring that can prevent code smells from occurring.

A decaying module can be detected by measuring the module decay index (MDI). MDI can also function as a quality indicator of non-smelly modules. We conducted empirical studies on decaying modules and found that 19% of the number of modules that are modified in each release becomes decaying modules. Additionally, compared to non-decaying modules, decaying modules have higher tendency to get decayed again in the future. Finally, we studied the use of a machine learning technique to predict decaying modules in the next release. We found that use of developers' context can improve the performance of the prediction model.

Chapter 7

Using Automated Impact Analysis Techniques to Improve Decaying Modules Prediction

Summary

A decaying module refers to a module whose quality is getting worse and is likely to become smelly in the future. The concept has been proposed to mitigate the problem that developers cannot track the progression of code smells and prevent them from occurring. To support developers in proactive refactoring process to prevent code smells, a prediction approach has been proposed to detect modules that are likely to become decaying modules in the next milestone. Our study in Chapter 6 has shown that modules that developers will modify as an estimation of developers' context can be used to improve the performance of the prediction model significantly. Nevertheless, it requires the developer who has perfect knowledge of locations of changes to manually specify such information to the system. To this end, in this study, we explore the use of automated impact analysis techniques to estimate the developers' context. Such techniques will enable developers to improve the performance of the decaying module prediction model without the need of perfect knowledge or manual input to the system. Furthermore, we conduct a study on the relationship between the accuracy of an impact analysis technique and its effect on improving decaying module prediction, as well as the future direction that should be explored.

This chapter is based on our paper "Can Automated Impact Analysis Techniques Help Predict Decaying Modules?" [99] published at the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME'19),

7.1 Introduction

Code smell is often used to represent an indicator of a design flaw or a problem in source code [1]. For instance, a class that has God Class code smell is considered as having too many functionalities and overly complex. Such problems are found to be related to software maintenance problems [19]. As a consequence, many approaches and tools have been proposed to detect code smells using different kinds of information such as source code metrics [3, 41] or historical information [24]. Nevertheless, one drawback of most tools is that they focus on a detect-and-remove strategy, which means that developers can only solve code smells after their source code become smelly. In other words, developers can neither track the progression of code smells or prevent them from occurring. To this end, in Chapter 6, we proposed the concept of a decaying module, which is a module whose quality is getting worse and is likely to become smelly in the future. This technique allows developers to prevent code smells from occurring.

In order to support developers' refactoring planning strategy, we also proposed a machine learning approach to predict modules that will decay in the future. The baseline uses source code quality as predictor variables and whether the given module will decay in the next release as a response variable. In addition to the baseline model, we also conducted a preliminary study on using developers' context (i.e., modules that developers are going to modify) as an additional predictor variable and found that such information can significantly improve the performance of the prediction model. We used a set of modules that developers are going to modify as a proxy to express developers' context. However, the preliminary study was conducted under the assumption that the developer who uses the system must have perfect knowledge of the locations of the changes. In addition, the developer has to specify such information into the system manually.

To bridge this gap, in this study, we explore the use of an alternative approach that can estimate the context of developers instead of relying on knowledge of developers. We leverage automated impact analysis techniques which refer to techniques that automatically identify a full set of modules that are likely to be affected by a particular change [33]. Such techniques use the information commonly available in a software development project, such as change descriptions, to predict modules that developers are going to modify. We conduct a study to verify whether the predicted modules can be used

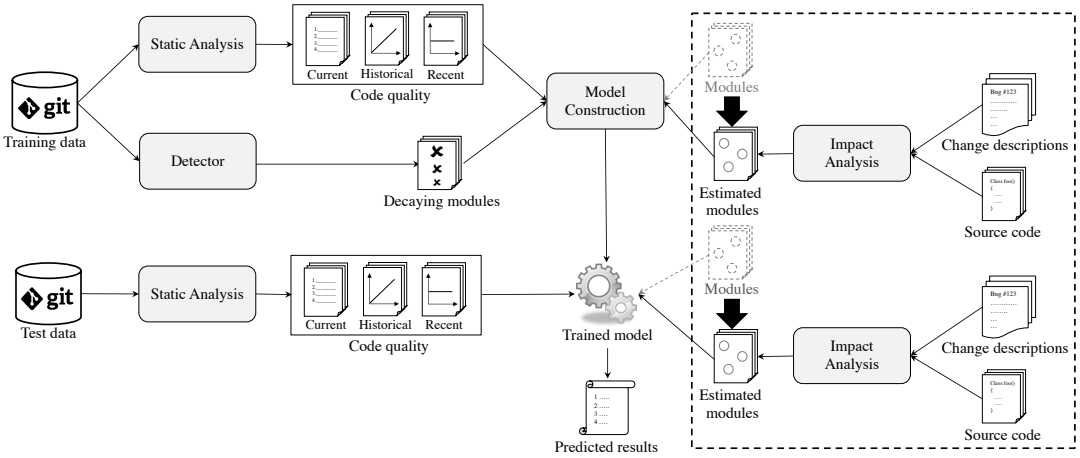


FIGURE 7.1: Decaying modules prediction approach.

as an estimation of developers' context, i.e., they can also be used to improve the prediction model performance of decaying modules. Such an approach will enable developers to improve the performance of the prediction model without the need for perfect knowledge of change locations nor the need for manual input to the system.

Moreover, while automated impact analysis techniques have been studied extensively in the literature [16], the approach is still far from being perfect. Although many attempts have been made to improve the accuracy of the approach, the relationship between the accuracy of an impact analysis technique and the performance of decaying module prediction remains unclear. Therefore, in this chapter, we study the relationship by artificially modifying the accuracy of impact analysis. The result can be used as empirical evidence to show how we can further improve the impact analysis technique in the future.

The main contributions of this study are twofold. First, we show that developers' context estimated by impact analysis techniques can also improve decaying module prediction performance. Second, we present an investigation on how we can improve the impact analysis technique to enhance the performance of decaying module prediction model further.

The remainder of this chapter is organized as follows. The next section summarizes the idea of a decaying module and its prediction. Section 7.3 presents our empirical studies. Section 7.4 presents threats to validity of this study. Section 7.5 concludes this chapter.

7.2 Decaying Module and Its Prediction

7.2.1 Decaying Module

In Chapter 6, the concept of a decaying module was proposed to mitigate the gap of most code smell detectors that warn developers about the quality problems only after they occur in the system, which did not allow developers to *prevent* the problems before. A decaying module was defined as a module that is getting closer to becoming smelly during a specified period. This would allow developers to handle modules before they become smelly.

In order to detect decaying modules, the Module Decay Index (MDI) was also proposed as an indicator of how close a module is to becoming code smell in Chapter 6. The MDI can be used in a situation of metric-based code smell detection that use multiple conditions to detect a code smell. Each condition is determined by whether a particular metric exceeds their corresponding thresholds. In this case, we can calculate MDI by averaging the ratio of a metric to its threshold of each condition. The MDI is range from 0 to 1. A higher MDI means that the module is closer to become a code smell. A module is considered a decaying module if its MDI has been increased from the previous release.

7.2.2 Decaying Modules Prediction

The idea of this approach is to use the information of the current release to predict modules that are going to decay in the next release. Figure 7.1 shows an overview of the baseline approach of our work in Chapter 6 and the modification we make in this study. The part outside the dashed box represents an overview of the baseline approach. The data is divided into training data and test data. Assuming that we want to predict if a given module will decay in release $n + 1$, the training data will be the source code of a set of releases $M = \{1, 2, \ldots, n - 1\}$ and the test data will be the source code of release n. First, for each release $m \in M$, we calculate three types of code quality of the source codes in training data to construct the classifier model. The three types of code quality are current code quality, historical code quality, and recent code quality trend. Then, we apply the detector to the source code of release $m + 1$ to identify decaying modules, i.e., modules that are getting closer to become smelly. Based on this information from all

releases in M, the predictor variables (code quality) and the response variable (whether the module gets decayed or not) are used to construct the classifier model. Then, we apply the same static analyzer to calculate the code quality of source code in release n and input to the model. Finally, the result of the model indicate modules that are likely to decay in release $n + 1$.

Additionally, the part that we change from the setting of our study in Chapter 6 is shown in the dashed box of Fig. 7.1. In Chapter 6, we showed that adding developers' context information, which is expressed as a set of modules that developers are going to modify, as an additional predictor variable can improve the performance of the prediction model significantly. However, as the primary purpose of our experiment in Chapter 6 was to obtain the highest improvement using the developers' context, we conducted a study under the condition that the developer has perfect knowledge of modules that will be modified. Furthermore, the developer has to specify such information into the system manually. To alleviate these conditions, in this study, we explore the use of an automated approach that does not require perfect knowledge of developers. Specifically, we focus on the use of information retrieval (IR)-based impact analysis technique to estimate developers' context. The underlying reason is that it requires a minimum amount of information to perform: only change descriptions and source code. Such information is commonly available in a software development project. The IR-based impact analysis takes the change descriptions that developers are going to implement and the source code to predict the locations that are going to be modified. The details will be explained in the next section.

7.3 Empirical Study

7.3.1 Research Questions

In this study, we aim at answering the following research questions:

RQ7.1: Can developers' context estimated by an IR-based impact analysis technique improve prediction performance?

RQ7.2: How can we further improve the performance of prediction model?

The details of each research question will be discussed later.

TABLE 7.1: Optimized Parameters of IR-Based Impact Analysis Techniques

Project	Technique	Cut point
Accumulo	BM25	20
Ambari	BM25	40
Derby	VSM	30
Hive	BM25	10

7.3.2 Experimental Setup

Data Collection

We use the same dataset as the baseline approach, which comprises of four open source projects: Accumulo, Ambari, Derby, and Hive. First, we obtain a list of issues of each project from their issue tracking systems[1]. Then, we extract summary, description, and the release that the issue was implemented. For each issue, we applied an impact analysis to generate locations that are likely to be changed to complete the issue. In this study, we focus on IR-based impact analysis because it requires minimum information to perform, i.e., it takes only the change descriptions and source code as inputs of the technique. The IR-based impact analysis works by calculating the textual similarity between an issue and source code. We consider the similarity score determined by the impact analysis technique as a probability that a particular module will be modified. Then, we calculate the Context Relevance Index (CRI), which was proposed in Chapter 4. The CRI of a module can be calculated by the summation of the similarity score in all issues that contains the module. The CRI represents the relevance of each module to the context of developers. A higher value of CRI means higher relevance to the context, i.e., more likely to be modified. We then create a new variable representing the CRI value. In order to calculate the CRI value, we need to specify two parameters: the technique used by IR-based impact analysis and the cut point when calculating CRI. The technique used by IR-based impact analysis determines how the data is represented and how the similarity is calculated while the cut point determines the number of modules used when calculating CRI. In this study, we adopt four fundamental IR-based impact analysis techniques which are often studied in impact analysis research: Vector

[1] `https://issues.apache.org/jira/secure/Dashboard.jspa`

Space Model (VSM), Latent Semantic Indexing (LSI), Latent Dirichlet allocation (LDA), and Okapi BM25 (BM25). The BM25 has been found to have the highest accuracy among IR-based approach [35]. For the cut point, we use 10, 20, 30, and 40, which are usually used in the previous research [33, 47]. For each project, we try all combination of each impact analysis technique and cut point, e.g., VSM with 10 cut point or VSM with 20 cut point. We then finally select the best combination of each project to use in this study. The underlying reason is that we want to simulate the real world setting where parameters of the technique are optimized for each project before utilizing the technique. The best combination of each project can be found in Table 7.1.

7.3.3 RQ7.1: Can developers' context estimated by an IR-based impact analysis technique improve prediction performance?

Motivation

As discussed earlier, we showed that developers' context (i.e., the modules that developers are going to modify) could be used to improve decaying module prediction performance. However, with the purpose of obtaining the upper bound performance of the model, the study was conducted based on the assumption of the perfect knowledge of developers. In other words, such an approach can be implemented only if the developer knows the location of the change perfectly. In addition, the developer has to input such information to the system manually. To this end, in this study, we propose an alternative approach that does not rely on perfect knowledge of developers. In other words, we use an automated approach that can predict or estimate the location where developers are going to make changes, i.e., an impact analysis technique. By using the results of automated impact analysis technique to represent developers' context, we suspect that it can also improve the performance of the model without the need of perfect knowledge from developers. However, since such techniques do not have perfect accuracy, it is sensible that such improvement is smaller than using the perfect knowledge of developers.

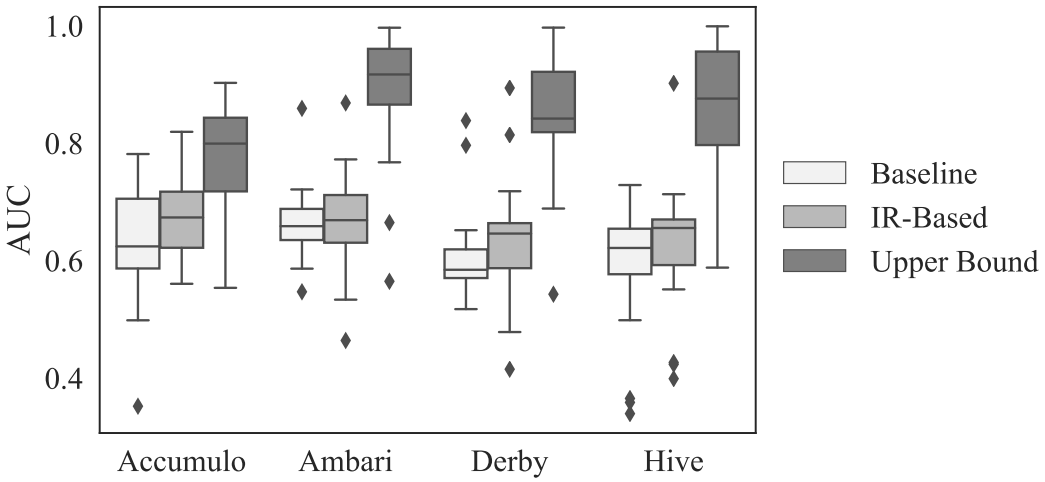


FIGURE 7.2: AUC values of baseline, IR-based models and the upper bound models.

Study Design

We compare the accuracy of the prediction model between the baseline and the model with CRI value. We conduct the Wilcoxon signed-rank test with the following null hypothesis to confirm if the results are statistically significant.

H_0: Developers' context estimated using IR-based impact analysis technique does *not improve* the performance of prediction model.

Therefore, an alternative hypothesis can be defined as:

H_a: Developers' context estimated using IR-based impact analysis technique *improves* the performance of prediction model.

In addition, we calculate Cliff's delta (d) as a measure of the magnitude of the improvement. The Cliff's delta is interpreted based on the threshold by Romano et al. [100]: negligible for $|d| < 0.147$, small for $|d| < 0.33$, medium for $|d| < 0.474$, and large for $|d| \geq 0.474$.

TABLE 7.2: Results of Wilcoxon Signed-Rank Tests and the Cliff's Delta Effect Size Tests

Project	p value	Cliff's delta
Accumulo	<0.001	0.293 (small)
Ambari	0.237	0.067 (negligible)
Derby	0.007	0.382 (medium)
Hive	0.002	0.218 (small)

Results and Discussion

Figure 7.2 shows the accuracy of the prediction model between the baseline model, the model using IR-based impact analysis techniques to estimate developers' context and the upper bound models. When comparing the accuracy between the baseline and IR-based models, we can observe that the IR-based model tends to have higher performance than the baseline model. Specifically, the median of the AUC values have been improved from 0.62 to 0.67 for Accumulo, from 0.66 to 0.67 for Ambari, from 0.58 to 0.65 for Derby, and from 0.62 to 0.66 for Hive.

Table 7.2 shows the results of Wilcoxon sign-rank tests and Cliff's delta effect size tests. The cells with p values less than 0.05 and Cliff's delta higher than 0.147 (not negligible) are highlighted in gray. The results of the Wilcoxon signed-rank test shows that the results are statistically significant at $\alpha = 0.05$ except for Ambari project. Therefore, for the projects other than Ambari, we can reject the null hypothesis and conclude that developers' context estimated by IR-based impact analysis technique can improve the performance of the prediction model. Furthermore, Cliff's delta values show that the result has a small effect for Accumulo and Hive, medium effect for Derby, and negligible effect for Ambari. So, we can see that the improvements are not negligible for all projects except for Ambari.

However, when we compare the improvement of the performance of the prediction model between IR-based and upper bound models, we can see that the improvements of IR-based models are much lower than the ones of the upper bound model. The result is as expected because the improvement of the upper bound model assumes perfect knowledge of developers, but the impact analysis technique relies on change descriptions to estimate the context. We can conclude that the IR-based impact analysis technique has

a potential of representing developers' context, although using only textual change descriptions and source code as inputs.

In conclusion, developers' context estimated by IR-based impact analysis technique can improve the performance of the decaying module prediction model.

7.3.4 RQ7.2: How can we further improve the performance of prediction model?

Motivation

As discussed in earlier sections, while the context estimated by existing IR-based impact analysis techniques can help improve the performance of prediction model, the improvements are still low comparing to the situation of perfect knowledge of developers. One reason for such small improvement may be the low accuracy of the IR-based impact analysis, i.e., the high number of false positives and the low number of true positives. If the impact analysis technique results in many false positives, they will become noises that may obstruct the prediction model instead of helping them identify decaying modules. Furthermore, if the number of true positives is low, the impact analysis techniques can predict only a part of the correct answers and fail to detect the rest and, therefore, provide insufficient information to the prediction model. To this end, many approaches have been proposed to improve the accuracy of impact analysis techniques such as combining IR-based approach with extra information [16]. Nevertheless, even the state-of-the-art approach is still far from being perfect. Although the research community has been working on improving the accuracy of impact analysis techniques, it is still unclear whether high accuracy impact analysis techniques can improve the decaying module prediction model. Thus, to obtain empirical evidence, we artificially tune the accuracy of impact analysis techniques and observe the relationship between the accuracy of an impact analysis technique and decaying module prediction model. We expect the result to be useful for the future direction of impact analysis research.

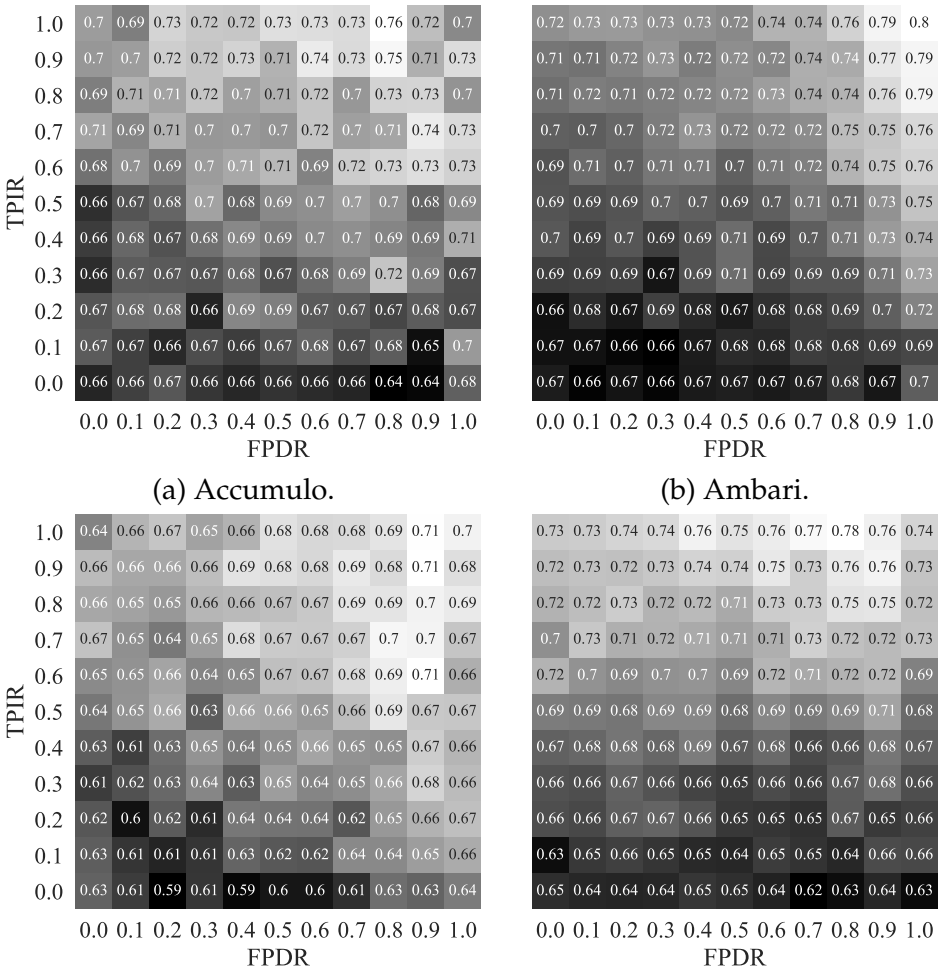
Study Design

We conducted an analysis under the assumption that mitigating the problems of high false positives and low true positives can improve the performance of the prediction model. We artificially modified the result of IR-based impact analysis technique in two steps, which are inspired by the task input generation approach of a feature location study [101]: First, we decrease the number of false positives by randomly removing false positives from the result. Second, we increase the number of true positives by randomly adding false negatives to the result. We refer to the ratio that we decrease the number of false positives as False Positive Decrement Ratio (FPDR) and to the ratio that we increase the number of true positives as True Positive Increment Ratio (TPIR). Both of the ratios are from 0.0 to 1.0 with the step of 0.1. After performing the modification, we recalculate the CRI and use it as an exploratory variable of the prediction model. Finally, we calculate the performance of each model for comparison.

Results and Discussion

Figure 7.3 represents the heat map of the AUC of the prediction model. The vertical axis represents the values of TPIR, while the horizontal axis represents the values of FPDR. Each cell represents the AUC value of each setting. The brighter color shows a higher AUC, while the darker color shows the lower AUC values. In general, we can observe that AUC values tend to have a higher value in the top right corner of the heat map (e.g., in Ambari project). This result suggests that the more we decrease the number of false positives, and the more we increase the number of true positives, the higher AUC values become. Moreover, when we observe the value with low TPIR, we can see that increasing FPDR does not significantly improve the AUC values. This may indicate that increasing the number of true positives should be given higher priority than decreasing the number of false positives.

Technically, decreasing the number of false positives may be accomplished by complimenting IR-based approach with other approaches such as dynamic analysis approach. For example, we can use execution trace to filter irrelevant modules from the result of IR-based approach, which may result in a lower number of false positives [33]. On the other hand, increasing the number of true positives can be done by combining the IR-based approach with a technique such as mining software repositories (MSR). For instance,



(a) Accumulo. (b) Ambari.

(c) Derby. (d) Hive.

FIGURE 7.3: Comparison of the prediction model performance at different TPIR and FPDR.

MSR approach can adopt association mining rules to detect modules that were often modified together in the past and use that information to detect the modules that may not be found by only IR-based approach [33]. While combining both techniques together have been shown to improve the accuracy of impact analysis techniques [33], one downside is that it requires extra information which may or may not be available depending on the projects.

To sum up, whereas improving the accuracy of the impact analysis techniques by decreasing the number of false positives and increasing the number of true positives can improve decaying modules prediction, we should give higher priority to increasing the number of true positives.

7.4 Threats to Validity

One significant threat to validity when using change descriptions to estimate developers' context lies in software repositories. For example, developers may make some modifications unrelated to any issue in the issue tracking system. In this situation, impact analysis techniques will fail to include such modification in the estimated list. In addition, as we rely on commit messages to identify the true positives of each issue, if developers do not put the details of the issue to the commit messages, our technique will fail to identify the true positives. We mitigated this threat by filtering the projects that have a high ratio of commits that include issue ID (higher than 80%) based on the list by Miura et al. [102]. This can ensure that most of the changes were related to the issues in the issue tracking system.

7.5 Conclusion

In this chapter, we explored the possibility of improving decaying modules prediction performance by using developers' context estimated by automated impact analysis techniques. We found that the context estimated by IR-based impact analysis techniques can improve the performance of the prediction model. We also discussed that the performance could be further improved by improving the accuracy of impact analysis technique such as decreasing the number of false positives and increasing the number of true positives of the results of impact analysis techniques.

Chapter 8

Conclusion

8.1 Summary of Contributions

In this dissertation, we explored approaches to support reactive and proactive refactoring using developers' context. Our approaches leverage information available in software development projects to identify locations where developers are likely to make changes. By combining the proposed approaches together, we can suggest source code locations that should be refactored by both removing existing code smells and preventing new code smells from occurring in the system. Our empirical studies with both open source software and professional developers demonstrate the effectiveness of our approaches.

The major contributions of this dissertation can be summarized as follows.

- **Factors that developers use to filter and prioritize code smells.** (Chapter 3)

  We conducted an experiment to examine how developers filter and prioritize code smells. We found that task relevance and smell severity are the most common factors that were considered during the code smell filtration process. For code smell prioritization process, we found that the most common factors are module importance and task relevance. These results can be used as a guideline for researchers and tool developers when developing new techniques.

- **Context-based code smell prioritization approach.** (Chapters 4 and 5)

  We presented a technique to prioritize code smells based on developers' context. The context is estimated by applying impact analysis techniques to a list of issues in an issue tracking system. Our technique

provides results as a list of code smells that are prioritized based on the relevance to developers' context. Such a result can support developers in prioritizing code smells to solve before making a modification. Our empirical studies show that our approach can provide more relevant results than the existing approach that use a factor unrelated to developers' context to prioritize code smells.

- **Decaying module.** (Chapter 6)

  We proposed the concept of a decaying module to use for supporting proactive refactoring. The decaying module is useful to warn developers of the modules that are going to become smelly modules. Our empirical studies found that decaying modules are more likely to get decayed again in the future compared to non-decaying modules. Such a result suggests that the decaying module is an effective indicator to control source code quality.

- **Decaying module prediction approach.** (Chapters 6 and 7)

  We presented a machine learning approach to predict modules that are going to get decayed in the next release. This approach can be used to facilitate developers when planning a refactoring strategy. While the baseline of the prediction model uses code quality metrics as predictor variables, we also found that developers' context can significantly improve the performance of the prediction model. Furthermore, we presented the use of developers' context estimated by impact analysis techniques to improve the prediction model.

8.2 Opportunities for Future Research

8.2.1 Code Smell Prioritization

In our code smell prioritization approach, the main factor that we used is the relevance to developers' context. Code smells with higher relevance to developers' context are given higher priority and put on the top of the list. In other words, we considered code smell prioritization as a mono-objective problem. However, there are many factors related to code smells that have not been studied. For example, the effort needed to fix code smells and the importance of each change descriptions. So, we can also consider the code

smell prioritization problem as a multi-objective problem by including such factors [103, 104]. Such direction remains as an opportunity for future research.

8.2.2 Defining Decaying Module of Other Types of Code Smell

In this dissertation, we presented the general definition of a decaying module and its related metrics. Such a definition can be applied to any types of code smells based on source code metrics. However, as a first step in exploring this direction, we have specifically defined the decaying module for only God Class code smell due to its simplicity. Thus, defining decaying modules for other types of code smell remains as opportunities for future research.

8.2.3 Improving Accuracy of Impact Analysis Techniques

As described herein, both code smell detection and decaying module prediction approach rely on impact analysis techniques to estimate developers' context. Therefore, the accuracy of impact analysis techniques significantly impacts the performance of both approaches. Consequently, improving the accuracy of impact analysis techniques can help improve the performance of our technique as well. Although many studies have been done to improve the impact analysis techniques such as combining with mining software repository or dynamic analysis approach, such approaches require extra information that may not always be available in software development projects. Therefore, developing approaches that have higher accuracy while limiting the information needed for the approach is a possible direction for future research.

Bibliography

[1] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] William C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.

[3] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[4] Yi Wang. "What motivate software engineers to refactor source code? evidences from professional developers". In: *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*. 2009, pp. 413–416.

[5] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "A field study of refactoring challenges and benefits". In: *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'12)*. 2012, 50:1–50:11.

[6] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. "Why We Refactor? Confessions of GitHub Contributors". In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 2016, pp. 858–870.

[7] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "An exploratory study on the relationship between changes and refactoring". In: *Proceedings of the 25th IEEE/ACM International Conference on Program Comprehension (ICPC'17)*. 2017, pp. 176–185.

[8] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. "Does Automated Refactoring Obviate Systematic Editing?" In: *Proceedings*

of the 37th International Conference on Software Engineering (ICSE'15). 2015, pp. 393–402.

[9] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. "An Experimental Investigation on the Innate Relationship between Quality and Refactoring". In: *Journal of Systems and Software* 107 (2015), pp. 1–14.

[10] Mik Kersten and Gail C. Murphy. "Using Task Context to Improve Programmer Productivity". In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'06)*. 2006, pp. 1–11.

[11] Min Zhang, Tracy Hall, and Nathan Baddoo. "Code bad smells: A review of current knowledge". In: *Journal of Software Maintenance and Evolution: research and practice* 23.3 (2011), pp. 179–202.

[12] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. "A review-based comparative study of bad smell detection tools". In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE'16)*. 2016, pp. 1–12.

[13] Tushar Sharma and Diomidis Spinellis. "A survey on software smells". In: *Journal of Systems and Software* 138 (2018), pp. 158–173.

[14] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 2013, pp. 672–681.

[15] Vaclav Rajlich. *Software Engineering: The Current Practice*. Chapman and Hall – CRC, 2011.

[16] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. "Feature Location in Source Code: A Taxonomy and Survey". In: *Journal of Software: Evolution and Process* 25.1 (2013), pp. 53–95.

[17] Aiko Yamashita and Leon Moonen. "Do Developers Care about Code Smells? An Exploratory Survey". In: *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*. 2013, pp. 242–251.

[18] Norihiro Yoshida, Tsubasa Saika, Eunjong Choi, Ali Ouni, and Katsuro Inoue. "Revisiting the relationship between code smells and refactoring". In: *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 2016, pp. 1–4.

[19] Aiko Yamashita and Leon Moonen. "Do Code Smells Reflect Important Maintainability Aspects?" In: *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM'12)*. 2012, pp. 306–315.

[20] Felienne Hermans and Efthimia Aivaloglou. "Do Code Smells Hamper Novice Programming? A Controlled Experiment on Scratch Programs". In: *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 2016, pp. 1–10.

[21] Michele Tufano, Fabio Palomba, Gabriele Bavota, and Rocco Oliveto. "When and Why Your Code Starts to Smell Bad". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. 2015, pp. 404–414.

[22] Tassio Vale and Iuri Santos Souza. "Influencing Factors on Code Smells and Software Maintainability: A Cross-Case Study". In: *Proceedings of the Second Workshop on Software Visualization, Evolution and Maintenance (VEM'14)*. 2014, pp. 1–8.

[23] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering* 36.1 (2010), pp. 20–36.

[24] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. "Detecting Bad Smells in Source Code using Change History Information". In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 2013, pp. 268–278.

[25] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. "A Textual-based Technique for Smell Detection". In: *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 2016, pp. 1–10.

[26] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mantyla. "Code Smell Detection: Towards a Machine Learning-based Approach". In: *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'03)*. 2013, pp. 396–399.

[27] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. "Detecting code smells using machine learning techniques: are we there yet?" In: *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. 2018, pp. 612–621.

[28] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. "Comparing heuristic and machine learning approaches for metric-based code smell detection". In: *Proceedings of the 27th International Conference on Program Comprehension (ICPC'19)*. 2019, pp. 93–104.

[29] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis". In: *Information and Software Technology* 108 (2019), pp. 115–138.

[30] Intooitus. *inFusion*. `http://www.intooitus.com/products/infusion`. (visited on 04/14/2015).

[31] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1983.

[32] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. "Indexing by Latent Semantic Analysis". In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407.

[33] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. "Integrated Impact Analysis for Managing Software Changes". In: *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 2012, pp. 430–440.

[34] Stephen Robertson, Hugo Zaragoza, et al. "The probabilistic relevance framework: BM25 and beyond". In: *Foundations and Trends® in Information Retrieval* 3.4 (2009), pp. 333–389.

[35] Md Masudur Rahman, Saikat Chakraborty, and Baishakhi Ray. "Poster: Which Similarity Metric to Use for Software Documents?: A Study on Information Retrieval Based Software Engineering Tasks". In: *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE'18)*. 2018, pp. 335–336.

[36] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "An Investigative Study on How Developers Filter and Prioritize Code Smells". In: *IEICE Transactions on Information and Systems* 101.7 (2018), pp. 1733–1742.

[37] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "How Do Developers Select and Prioritize Code Smells? A Preliminary Study". In: *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 2017, pp. 484–488.

[38] Radu Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". In: *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*. 2004, pp. 350–359.

[39] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. "JDeodorant: Identification and Removal of Type-checking Bad Smells". In: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. 2008, pp. 329–331.

[40] Matthew James Munro. "Product Metrics for Automatic Identification of 'Bad Smell' Design Problems in Java Source-Code". In: *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. 2005, pp. 1–9.

[41] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. "Towards a Prioritization of Code Debt : A Code Smell Intensity Index". In: *Proceedings of the IEEE Seventh International Workshop on Managing Technical Debt (MTD'15)*. 2015, pp. 16–24.

[42] Francesca Arcelli Fontana, Vincenzo Ferme, and Marco Zanoni. "Poster: Filtering Code Smells Detection Results". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. 2015, pp. 803–804.

[43] Radu Marinescu. "Assessing Technical Debt by Identifying Design Flaws in Software Systems". In: *IBM Journal of Research and Development* 56.5 (2012), 9:1–9:13.

[44] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Context-based code smells prioritization for prefactoring". In: *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*. 2016, pp. 1–10.

[45] Roberta Arcoverde, Everton Guimaraes, Isela Macia, Alessandro Garcia, and Yuanfang Cai. "Prioritization of Code Anomalies Based on Architecture Sensitiveness". In: *Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES'13)*. 2013, pp. 69–78.

[46] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. "An Approach to Prioritize Code Smells for Refactoring". In: *Automated Software Engineering* 23.3 (2016), pp. 501–532.

[47] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Context-Based Approach to Prioritize Code Smells for Prefactoring". In: *Journal of Software: Evolution and Process* 30.6:e1886 (2018), pp. 1–24.

[48] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness". In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275.

[49] Latifa Guerrouj, Zeinab Kermansaravi, Venera Arnaoudova, Benjamin CM Fung, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc.

"Investigating the relation between lexical smells and change-and fault-proneness: an empirical study". In: *Software Quality Journal* 25.3 (2017), pp. 641–670.

[50] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension". In: *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*. 2011, pp. 181–190.

[51] Aiko Yamashita and Leon Moonen. "Exploring the impact of inter-smell relations on software maintainability: An empirical study". In: *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 2013, pp. 682–691.

[52] Aiko Yamashita and Leon Moonen. "To what extent can maintenance problems be predicted by code smell detection?–An empirical study". In: *Information and Software Technology* 55.12 (2013), pp. 2223–2242.

[53] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. "Quantifying the effect of code smells on maintenance effort". In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1144–1156.

[54] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. "Do Code Smells Impact the Effort of Different Maintenance Programming Activities?" In: *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. Vol. 1. 2016, pp. 393–402.

[55] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. "Do they really smell bad? a study on developers' perception of bad code smells". In: *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. 2014, pp. 101–110.

[56] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. "Understanding the longevity of code smells: preliminary results of an

explanatory survey". In: *Proceedings of the 4th Workshop on Refactoring Tools (WRT'11)*. 2011, pp. 33–36.

[57] Ralph Peters and Andy Zaidman. "Evaluating the lifespan of code smells using software repository mining". In: *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*. 2012, pp. 411–416.

[58] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Revisiting Context-based Code Smells Prioritization: On Supporting Referred Context". In: *Proceedings of the Ninth International Workshop on Managing Technical Debt (MTD'17)*. 2017, 3:1–3:5.

[59] Daniel Ratiu, Stephane Ducasse, Tudor Girba, and Radu Marinescu. "Using History Information to Improve Design Flaws Detection". In: *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)*. 2004, pp. 223–232.

[60] Anselm Strauss and Juliet Corbin. *Basics of qualitative research: Procedures and techniques for developing grounded theory*. Thousand Oaks, CA: Sage, 1998.

[61] Rashina Hoda, James Noble, and Stuart Marshall. "Using grounded theory to study the human aspects of software engineering". In: *Proceedings of the Second Workshop on Human Aspects of Software Engineering (HAoSE'10)*. 2010, 5:1–5:2.

[62] Shawn A Bohner and Robert S Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[63] Alexander Kohan, Mitsuharu Yamamoto, and Cyrille Artho. "Automated Dataset Construction from Web Resources with Tool Kayur". In: *Proceedings of the Fourth International Symposium on Computing and Networking (CANDAR'06)*. 2016, pp. 98–104.

[64] Bogdan Dit, Michael Wagner, Shasha Wen, Weilin Wang, Mario Linares-Vásquez, Denys Poshyvanyk, and Huzefa Kagdi. "ImpactMiner: A Tool for Change Impact Analysis". In: *Companion Proceedings of the*

36th International Conference on Software Engineering (ICSE'14). 2014, pp. 540–543.

[65] Bogdan Dit, Andrew Holtzhauer, Denys Poshyvanyk, and Huzefa Kagdi. "A Dataset from Change History to Support Evaluation of Software Maintenance Tasks". In: *Proceedings of the Tenth Working Conference on Mining Software Repositories (MSR'13)*. 2013, pp. 131–134.

[66] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. "TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions". In: *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 2012, pp. 1375–1378.

[67] Bogdan Dit, Evan Moritz, and Denys Poshyvanyk. "A TraceLab-based Solution for Creating, Conducting, and Sharing Feature Location Experiments". In: *Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC'12)*. 2012, pp. 203–208.

[68] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., 1998.

[69] Arthur J Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[70] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2007.

[71] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 2000.

[72] Kalervo Järvelin and Jaana Kekäläinen. "Cumulated Gain-based Evaluation of IR Techniques". In: *ACM Transactions on Information Systems* 20.4 (2002), pp. 422–446.

[73] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. 1st ed. Addison-Wesley Publishing Company, 2009.

[74] Emerson Murphy-Hill and Andrew P Black. "Refactoring tools: Fitness for purpose". In: *IEEE software* 25.5 (2008), pp. 38–44.

[75] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. "Preliminary catalogue of anti-pattern and code smell false positives". In: *Poznan University of Technology, Technical Report RA-5/15* (2015), pp. 1–28.

[76] Francesca Arcelli Fontana, Jens Dietrich, Bartosz Walter, Aiko Yamashita, and Marco Zanoni. "Antipattern and code smell false positives: Preliminary conceptualization and classification". In: *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. 2016, pp. 609–613.

[77] Mik Kersten and Gail C. Murphy. "Mylar: a degree-of-interest model for IDEs". In: *Proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD'05)*. 2005, pp. 159–168.

[78] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?" In: *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. 2012, pp. 79–103.

[79] Stanislav Negara. "Towards a Change-Oriented Programming Environment". PhD thesis. University of Illinois, 2013.

[80] Ethan Zhang and Yi Zhang. "Average Precision". In: *Encyclopedia of Database Systems*. Springer, 2009, pp. 192–193.

[81] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. "A Bayesian Approach for the Detection of Code and Design Smells". In: *Proceedings of the Ninth International Conference on Quality Software (QSIC'09)*. 2009, pp. 305–314.

[82] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "Numerical Signatures of Antipatterns: An Approach Based on B-Splines". In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*. 2010, pp. 248–251.

[83] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. "Code-smell Detection as a Bilevel Problem". In: *ACM Transactions on Software Engineering and Methodology* 24.1 (2014), 6:1–6:44.

[84] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. "Comparing and Experimenting Machine Learning Techniques for Code Smell Detection". In: *Empirical Software Engineering* 21.3 (2016), pp. 1143–1191.

[85] Zadia Codabux and Byron J. Williams. "Technical Debt Prioritization Using Predictive Analytics". In: *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE'16)*. 2016, pp. 704–706.

[86] Shinpei Hayashi, Motoshi Saeki, and Masahito Kurihara. "Supporting Refactoring Activities Using Histories of Program Modification". In: *IEICE Transactions on Information and Systems* E89-D.4 (2006), pp. 1403–1412.

[87] Hui Liu, Xue Guo, and Weizhong Shao. "Monitor-Based Instant Software Refactoring". In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1112–1126.

[88] Rodrigo Morales, Zéphyrin Soh, Foutse Khomh, Giuliano Antoniol, and Francisco Chicano. "On the use of Developers' Context for Automatic Refactoring of Software Anti-patterns". In: *Journal of Systems and Software* 128 (2017), pp. 236–251.

[89] Emerson Murphy-Hill and Andrew P. Black. "Seven Habits of a Highly Effective Smell Detector". In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering (RSSE'08)*. 2008, pp. 36–40.

[90] Akihiro Yamamori, Anders Mikael Hagward, and Takashi Kobayashi. "Can Developers' Interaction Data Improve Change Recommendation?" In: *Proceedings of the 41st Annual IEEE Computer Software and Applications Conference (COMPSAC'17)*. 2017, pp. 128–137.

[91] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Toward Proactive Refactoring: An Exploratory Study on Decaying Modules". In: *Proceedings of the third International Workshop on Refactoring (IWoR'19)*. 2019, pp. 39–46.

[92] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. "When and why your code starts to smell bad (and whether the smells go away)". In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1063–1088.

[93] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. "A validation of object-oriented design metrics as quality indicators". In: *IEEE Transactions on software engineering* 22.10 (1996), pp. 751–761.

[94] Jevgenija Pantiuchina, Gabriele Bavota, Michele Tufano, and Denys Poshyvanyk. "Towards just-in-time refactoring recommenders". In: *Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18)*. 2018, pp. 312–315.

[95] Leo Breiman. "Random forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[96] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.

[97] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. "Does code decay? Assessing the evidence from change management data". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12.

[98] Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "A Preliminary Study on Using Code Smells to Improve Bug Localization". In: *Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC'18)*. 2018, pp. 324–327.

[99] Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Can Automated Impact Analysis Techniques Help Predict Decaying Modules?" In: *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*. to appear. 2019.

[100] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys". In: *Annual meeting of the Florida Association of Institutional Research*. 2006, pp. 1–33.

[101] Takashi Ishio, Shinpei Hayashi, Hiroshi Kazato, and Tsuyoshi Oshima. "On the Effectiveness of Accuracy of Automated Feature Location Technique". In: *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13)*. 2013, pp. 381–390.

[102] Keisuke Miura, Shane McIntosh, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. "The impact of task granularity on co-evolution analyses". In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (EMSE'16)*. 2016, pp. 1–10.

[103] Thainá Mariani and Silvia Regina Vergilio. "A systematic review on search-based refactoring". In: *Information and Software Technology* 83 (2017), pp. 14–34.

[104] Michael Mohan and Des Greer. "A survey of search-based refactoring for software maintenance". In: *Journal of Software Engineering Research and Development* 6.1 (2018), pp. 1–52.

Publications and Awards

Journal Papers

1. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "An Investigative Study on How Developers Select and Prioritize Code Smells". In: *IEICE Transactions on Information and Systems*, 101.7 (2018), pp. 1733–1742.

2. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Context-based approach to prioritize code smells for prefactoring". In: *Journal of Software: Evolution and Process*, 30.6:e1886 (2018), pp. 1–24.

International Conference/Workshop Papers

1. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Can Automated Impact Analysis Techniques Help Predict Decaying Modules?". In: *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME'19)*, 2019, Cleveland, Ohio, USA. (to appear)

2. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Toward Proactive Refactoring: An Exploratory on Decaying Modules". In: *Proceedings of the Third International Workshop on Refactoring (IWoR'19)*, 2019, pp. 39–46, Montreal, Canada.

3. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "How Do Developers Select and Prioritize Code Smells? A Preliminary Study". In: *Proceedings of the 33th IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 484–488, Shanghai, China.

4. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Revisiting Context-based Code Smells Prioritization: On Supporting Referred

Context". In: *Proceedings of the Ninth International Workshop on Managing Technical Debt (MTD'17)*, 2017, 3:1–3:5, Cologne, Germany.

5. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Context-based Code Smells Prioritization for Prefactoring". In: *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC'16)*, 2016, pp. 1–10, Austin, Texas, USA.

Other Publications

1. Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Using Code Smells to Improve Information Retrieval-Based Bug Localization". In: *IPSJ Journal*, 60.4 (2019), pp. 1040–1050. **(Specially Selected Paper Award)**

2. Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "A Preliminary Study on Using Code Smells to Improve Bug Localization". In: *Proceedings of the 26th IEEE International Conference on Program Comprehension (ICPC'18)*, 2018, pp. 1–10, Gothenburg, Sweden.

3. Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Using Code Smell Severity to Improve IR-Based Bug Localization". In: *IPSJ Technical Reports*, 2018-SE-198.16 (2018) pp. 1–8, Tokyo, Japan. **(Student Research Award) (IPSJ Computer Science Research Award for Young Scientists)**

4. Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. "Toward Prioritizing Code Smell Detection Results for Prefactoring". In: *IEICE Technical Reports*, 115.153 (2015), pp. 33–38, Hokkaido, Japan. **(Research Award)**

Awards

1. Specially Selected Paper Award, IPSJ, 2019

2. Tejima Research Award, Tokyo Institute of Technology, 2018

3. Research Award, Software Science Research Society, IEICE, 2015