UNDERSTANDING THE EVOLUTION OF CODE SMELLS BY OBSERVING SMELL CLUSTERS

AHMAD TAHMID
Master of Science in Software Engineering
Institute of Information Technology, University of Dhaka
Class Roll: MSSE 0306
Registration Number: 2010-815-391
Session: 2014-15

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Master of Science in Software Engineering

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

©Ahmad Tahmid - Institute of Information Technology, University of Dhaka, 2016

UNDERSTANDING THE EVOLUTION OF CODE SMELLS BY OBSERVING SMELL CLUSTERS

AHMAD TAHMID

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Mohd. Zulfiquar Hafiz

Committee Member: Dr. Mohammad Shoyaib

Committee Member: Rayhanur Rahman

Committee Member: Dr. Muhammad Mahbub Alam

Student: Ahmad Tahmid

To *Hamida Begum*
for being my mother

Abstract

Code smells are more likely to stay inter-connected in software rather than remaining as a single instance. These code smell clusters create maintainability issues in evolving software. This paper aims to understand the evolution of the code smells in software, by analyzing the behavior of these clusters such as size, number and connectivity. For this, the clusters are first identified and then these characteristics are observed. The identification of code smell clusters is performed in three steps - detection of code smells (God Class, Long Method, Feature Envy, Type Checking) using smell detection tools, extraction of their relationships by analyzing the source code architecture, and generation of graphs from the identified smells and their relationships, that finally reveals the smelly clusters. This analysis was executed on JUnit as a case study, and four important cluster behaviors were reported.

Acknowledgments

I would like to thank my supervisor Associate Professor Dr. Kazi Muheymin-Us-Sakib for his support and guidance during my candidature. His constant encouragement was one of the main driving forces that kept me motivated. I would like to thank him for all his help with the research directions and experimentation. All his comments and suggestions were invaluable. His very demanding nature of supervising as well as pushing me to the limits have helped me grow as a researcher.

I would also like to thank my committee members for serving even at hardship. I also want to thank them for letting my defense be an enjoyable moment and for their brilliant comments and suggestions.

Also I would like to thank other faculty members for their participation and constructive feedback in the production of the thesis.

I am thankful to all other staffs at IIT, University of Dhaka for creating a pleasant work environment.

I am expressing ever gratefulness to all my fellow classmates whose advice, feedback and cooperation is truly incomparable.

Finally, I am indebted to my parents for the many years of hard work and sacrifices they have made to support me. Without them, this thesis would never have started.

Publication

"Understanding the Evolution of Code Smells by Observing Smell Clusters," *In Proceedings of the 3rd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP) co-located with the 23rd IEEE International Conference on Software Analysis, Evolution, and Re-engineering (SANER)*, Osaka, Japan, March 15, 2016.

Contents

List of Figures

Chapter 1

Introduction

Abstract In a large software, a significant percentage of code smells are inter-connected and co-occurred [1]. Studies, such as [2], showed that a combination of code smells is more difficult to manage compared to a single instance of code smell. In this study, these inter-related smelly code components are referred as smell clusters. These smell clusters can have a serious impact on the manageability and overall quality of software, specially if the software is evolving. This is because, as a system grows older, instances of smells in it's source code go through a complex process of evolution [3] and often lead to bigger architectural problems known as anti-patterns [4].

1.1 Key Terms

Before discussing the motivation and target of this study, few key terms are described, to make it easier for the readers who are new to Code Smell research domain.

Code Smells are common coding practices which make code difficult to understand for other developers [5]. Code smells do not hamper programs performance or accuracy, but decrease understandability. In this thesis, the term 'code smell', 'smell'

Before Refactoring

```
//Smelly Method
public void updateUser(String name, int age, String address, String email, String phone)
{
    //Code for updating a user
}
```

After Refactoring

```
//Smell-Free Method
public void updateUser(User newUser)
{
    //Code for updating a user
}

class User
{
    String name;
    int age;
    String address;
    String email;
    String phone;
}
```

Figure 1.1: Code Smell and Refactoring

and 'bad smell' are used interchangeably.

Smelly Components are the methods and classes that contain at least one code smell [6]. There are two type of smelly component - 'smelly class' and 'smelly method'.

Refactoring means rewriting or changing code to remove code smells from it [5].

In Figure 1.1 (Top), the method 'updateUser' takes too many parameters to perform perform some tasks. Although this function works correctly, it is not easy to understand it at a glance because of the long list of parameters. This practice is known as 'Long Parameter List' smell. 'addUser' is a 'smelly method' as it contains 'Long Parameter List' smell, and therefore, it should be refactored. In Figure 1.1 (Bottom), a refactored version of the code is presented, in which, 'addUser' is smell-free.

Smell Clusters are sets of architecturally connected smelly components. For example, consider a smelly class C1, that contains a smelly method M1. So, M1 and C1

are architecturally connected. Now consider a non-smelly class C2 contains a smelly method M2. If M1 calls the method M2, then, C1, M1, and M2 is architecturally connected. Therefore, C1, M1 and M2 is a smell cluster.

1.2 Motivation

Improving Code Quality. To keep pace with the ever-changing technology, a software has to undergo a lot of changes, updates and bug-fixes. Changing or fixing a software takes a lot of development time and effort if it's code is not easily understandable. Therefore, quality of a code depends on its understandability and maintainability. However, in real-life software development, developers often ignore the quality of code to optimize time and effort. This is because, they have to handle other important issues of software development such as, performance, deadline, resource management, etc. As a result, often a degradation of code quality is encountered.

Minimizing Code Smells. The common symptoms which indicate degradation in code quality, are known as code smells. In real-life software development, it not always possible to ensure quality of code, as a result, introduction of code smells in imminent. According Chatzigeorgiou *et al.* [7], code smell is a recurring problem, and cannot be eradicated completely. As code smell problem cannot be solved for good, developers must refactor as many smells as possible and keep code maintainable. To identify the most dangerous smells and refactor those, it is important to understand the behavior of code smell.

Understanding the Behavior of Code Smells. During the last two decade, many researchers concentrated on understanding the behavior of code smells [8, 3]. The early researches on code smell tried to identify the most harmful smells [9, 10].

However, it is found that, a no single smell carries as much threat to maintainability as a combination of smells [2]. Therefore, researchers concentrated on correlation and co-occurrence on smells [11, 1]. They found that smells tend be interconnected and form clusters of smells in software. This cluster of bad smells create severe understandability and maintainability issues.

Understanding the Evolution of Smell Clusters. In spite of the problems arose by the smell clusters in the evolution of software, it has not got enough focus in research due to the complex connectivity of the smells. Although researchers have focused on evolution of individual smells, to the best of authors' knowledge no study have been conducted on the evolution of smell clusters. Therefore, to achieve greater knowledge about software maintainability, it is needed to observe smell clusters, and understand how these clusters evolve in different versions of the software. That's why the ultimate aim of this study is - *'Understanding the Evolution of Smell Clusters'*

By achieving this goal, a better understanding of code smells can be obtained. This will help to minimize code smells and ultimately improve code quality.

1.3 Research Questions

The target of this study is to understand how smelly clusters evolve with time. Three research questions have been identified which can lead to this Target.

RQ1: How does the architectural connectivity of smelly components evolve with time?

The first research question of this study, aims to understand how architectural connection between smelly components change over time. In the literature of code smells, it is seen that, number of smelly components in a software increase steadily with time. Therefore, it can be assumed that, architectural connection between smelly

components will also increase from one version to the next. The target is to see whether number of architectural connections in version n increases in version n+1.

RQ2: How does the existence of code smell clusters change over time?

This research question aims to understand how the smell clusters existing in a specific version of software change in future versions of it. As the number of code smells in a software increases with time, it can be assumed that, these smells will create more clusters in later versions of software. Therefore, the target is to observe whether the number of smelly clusters in version n of a software increase in version n+1.

RQ3: Do all the smell clusters show similar pattern of evolution?

In this research question, the aim is to investigate, whether the smell clusters show any specific pattern of evolution. It is known that, number of smelly components rises as a software grows older. If it is assumed that, architectural connections between smelly components also increase during this process, it is highly possible that, larger clusters of smells will form. These large clusters might ultimately create a giant cluster. Therefore, to answer this research question, the target is to investigate whether smell clusters ultimately tend to create a giant cluster of smells. It is described in the literature that, a combination of smells makes code relatively less maintainable compared to a single smell. As the predicted giant cluster will holds a lot of smells, it must be very difficult to manage. Therefore, the assumption is, more and more new smells will be introduced into this cluster, and smell removal rate will be very low. It will be investigated if similar situation is found in real-life systems.

This study is guided by these research questions. The goal is to find out whether the assumption derived from the research question are accurate or not.

1.4 Contribution

- **FUESC:** A novel framework 'FUESC - Framework for Understanding Evolution of Smell Clusters' is proposed in this study to investigate evolution of smell clusters in a system. This framework presents a technique that observes consecutive versions of a software and extracts smell cluster properties such as change of cluster size, change of connectivity in clusters, evolution of clusters, etc.

- **SCIT:** A java tool 'SCIT' is presented that implements 'FUESC' to understand clustering behavior in real-life software. This tool analyzes different version of java projects and identify smelly components, their connectivity, and finally extracts smell clusters. By comparing smell clusters in different version of a software, it can present evolution pattern of these clusters.

- **Findings:** To understand evolution pattern of smell clusters, using SCIT tool, 25 version of 3 open-source java projects (JUnit, Mockito and Commons-lang) are examined. The finding from the case study is, smelly components tend to architecturally connect with other smelly components in software, create smells clusters, and eventually form a giant 'Mega Cluster' in which, smell initiation is more likely to happen than smell elimination.

A case study was conducted to understand how the relationships between smelly components of a source code change with the evolution of software. JUnit was selected for the study. For 10 different versions of JUnit smell clusters were detected. By comparing the clustering properties such as cluster size, cluster count, connectivity etc. for all the versions few definitive characteristics were identified; that are - number of connected code smell clusters increases steadily with time, smell clusters can be categorized in three types (mega cluster, single-node cluster, and small cluster), smelly

components tend to create a mega cluster, and the size of the mega cluster increases steadily with time.

1.5 Organization of the Thesis

The rest of this thesis is organized as follows.

- **Chapter 2: Literature of Code Smells** In this chapter, a brief about researches targeting code smells, and it's impact and evolution is presented
- **Chapter 3: FUESC: Framework for Understanding the Evolution of Smell Clusters** A novel technique is proposed using which multiple releases of a software can be analyzed and clustering behavior of smells can be identified.
- **Chapter 4: SCIT: Smell Cluster Inspection Tool** A tool 'SCIT' is presented in this chapter. A case study is performed using this tool. The result of the case study is also demonstrated in this chapter.
- **Chapter 5: Result Analysis** This chapter discusses the results of the experiment and explains how those results answers the research questions of this study.
- **Chapter 6: Conclusion** In this concluding chapter, a brief of the whole work is summarized and future directions are described.

Chapter 2

Literature of Code Smell

The features of code that decrease code quality and make it less maintainable, are known as code smells [5]. Code smells make code difficult to understand or change, and thus, make it difficult for developers to update, fix or enhance a software [12, 13]. Therefore, it is very important to eradicate smells for maintaining clean and understandable code [14]. However, in real-life situation, it is not always feasible for developers to maintain smell-free code as they have to handle other problems of software development too such as resource limitation, pressure of deadline, software performance, etc. Therefore, it is very important to understand how code smells behave in a software. During the last two decades many researchers investigated code smell, and its impact, relation and evolution. The literature shows that smells residing in a software, degrade software quality, create complex relations, and influence the evolution of the whole system.

In this chapter, first, definition of different code smells and taxonomy of these are discussed. After that well-known techniques to detect code smells are described. Then, it is presented how code smells degrade software quality, and finally the relations between code smells and evolution of those are described in separate sections.

2.1 Code Smells

Undesired design flaws, known as **code smells (or smells or bad smells)**, are widely considered as indicators of decrease in software quality [15]. In 1999, Martin Folwer first identified a set of common symptoms in code that are threat to software quality and introduced the term 'Code Smell' to denote those [5]. According to his definition, code smells are surface indications of bad design practices implemented by developers. Although code smells do not interfere with the functionality, accuracy or performance of a software, it makes a code difficult to understand [16]. For instance, a long method with thousands of statements might provide accurate outputs in real-time, but it certainly is not easy for a developer to understand it. In the same way, use of structures and primitive data types instead of objects might speed-up a process, but it will make it less updatable. Therefore, code smells do not bother the users of a software, but those are big headaches for the developers.

History of code smell research. Research on code smells has a relatively short history. Before the concept of 'Code Smell' was introduced, not many researches were conducted to estimate quality of object oriented code. Chidamber *et al.* [17], in 1994, first proposed a metric based approach to estimate object oriented code quality. In the same year, Martin *et al.* [18] conducted a study to measure aspects of object oriented code by using code metrics. The first book written on bad object oriented design practices titled as 'Pitfalls of object oriented development' was published in 1995 by Webster [19]. However, in this book, the authors focused only on common violations of object oriented concepts. Later in 1999, Fowler defined 23 symptoms in code that indicate degradation in code quality [5] and named those as 'Code Smells'. In the same year, Beck *et al.* published refined definitions of 22 bad smells that are commonly found in methods or classes.

What causes code smells? According to Zazworka *et al*, inexperience and

bad design practices have highest contribution in code smell's initiation in software [20]. Other factors that contribute to introduction of smells are - unplanned software structure, constant change of requirements, change of developers, shortcut implementation, etc. [21]. Chatzigeorgiou investigated how code smells are introduced to a system, and found that most of the smells are initiated when new methods or classes are added to source code [7]. From these studies, it can be seen why and how smells are generated, which will help to understand the behavior of code smells.

Code smells can not be removed completely. In ideal situation, all smells should be refactored, and code should be hundred percent smell free. However, in software development world, the situation it is not always ideal as there exists resource limitation, constant pressure of deadline and lack of experienced developers. To understand this situation, Peter *et al.* observed developers of seven open-source systems and found that, developers are aware of bad smells, though they do not often try to remove those, given the low refactoring activity [3]. It indicates that, in real-life software development, it is not always feasible to maintain smell free code [3].

Categories of code smell researches. As code smells cannot be completely eradicated, researchers focused on different aspects of code smells such as taxonomy, detection, impact, evolution, etc. In the first few studies, researchers tried to define and classify smells [21, 22] which were followed by researches on smell detection processes [23, 24]. The more recent studies tend to aim at impact, relation and evolution of code smells [9, 8, 25]. In the next five sections, these five categories of code smell researches are described.

2.2 Taxonomy of Smells

Early researches on code smells mostly concentrated on definitions and types of code smells [21, 22]. After Beck *et al.* defined 22 code smells in 1999, many researchers grouped smells based on occurrence, influence, refactorability, etc. In 2003, M Mantyla *et al.* first published a complete taxonomy of all code smells based on the design problems those indicate [22]. Wake's classification, which was published in 2004 was based on refactorability [26]. Later, in 2006, R Marticorena published an extended list of code smells divided in six groups [27]. Those six groups are Bloaters, Object Oriented Abusers, Change Preventers, Dispensables, Couplers and Others.

All 22 smells in those groups are described below -

Bloaters: When code segments such as classes and method get extremely large in size and become highly unmanageable, those are called bloaters. Bloaters residing in code, increase in size as time elapses (unless anybody makes an effort to eradicate those) and create giant blocks of code that are very difficult to work with. Smells in this category are described below.

- **Data Clumps** A Data Clump is a set of variables that are always used together. Identical copies of a data clump can be found in many different places in code. Data clumps are results of poor program structure and 'copy paste programming' [5].

- **Large Class or God Class** A God Class is a class that has become extremely large in size, controls a lot of other classes and performs too many tasks. A God Class is very hard to understand or maintain because of its size and complexity [5].

- **Long Method** Long method or brain method is a method that performs too many duties instead of one. It is large in size and has a high amount of complixity in it [5].

- **Long Parameter List** A method that takes too many parameters every time it is called, is said to have Long Parameter List smell. Existence of this smell indicates existence of data clumps in code.

- **Primitive Obsession** When primitive data types are used instead of advanced types or objects, it is called primitive obsession.

O-O Abusers: O-O abusers are code smells which are generated when object

oriented concepts are violated. Existence of these smells indicate problems in encapsulation, inheritance and polymorphism.

- **Alternative Classes with Different Interfaces** When developers use concrete alternative classes instead of using interfaces, it is considered as a smell. This practice limits the use of inheritance in object oriented code.

- **Refused Bequest** Sometimes a class is forced to implement a super class just for ease of implementation. This improper use of inheritance is considered as a bad smell.

- **Switch Statements** Long conditional statement indicates that a code is more structural than object oriented. If conditional switch statements are used instead of object checking, it is considered as a code smell.

- **Temporary Field** Developers often use temporary fields to stores values of an object instead of creating a property inside of that object. This practice is known as Temporary Field smell.

Change Preventers: Code smells that decrease changeability, updatability or fixability of a code, are grouped in Change Preventers group. Presence of these smell, makes it tough for developers to perform maintenance tasks.

- **Divergent Change** If a developer have to perform a series of changes in many places to implement a change, it means there is severe violation of encapsulation and this situation is known as divergent change smell.

- **Parallel Inheritance Hierarchies** When developer implement multiple hierarchies for single structure, it is considered as a design problem. This situation is a result of unplanned programming.

- **Shotgun Surgery** When developers implement same features in different ways though out a software, it creates severe maintainability issues. This practice is known as shortgun surgery smell.

Dispensables: Dispensables are group of those smells, which are unnecessary to code and should be removed by refactoring.

- **Data Class** Data classes are containers that hold data of an object but do not perform any operation on those. This type of object implementation are violations of object oriented concepts and should be removed.

- **Duplicate Code** Duplicate code are results of copy paste programming. Code duplicity creates confusion while performing maintenance tasks, hence should be removed or unified.

- **Lazy Class** Lazy classes are classes that holds properties, but rely on other classes to perform operations on those properties. Instances of these type of classes are known as bad smells.

- **Speculative Generality** When code is implemented on unnecessarily granular level, it is called speculative granularity smell.

Couplers: Smell that initiates unnecessary intimacy between classes and methods are grouped in Couplers group.

- **Feature Envy** If a class has excessive intimacy with a method of any other class, then that method should probably be situated inside the calling class. This excessively enviousness of a class towards a non local method in known as feature envy.

- **Inappropriate Intimacy** If a class excessively uses properties and methods of another class, those classes are said to have inappropriate intimacy.

- **Message Chains** When sending a message from one method to other method requires to go through other methods, it is called message chain. This situation is considered as a code smell.

- **Middle Man** If a class has to use some other class to access a third class, it is called Middle Man problem. This problem indicates bad design practices.

Not Defined: There exist two smells that cannot be grouped in any category. Beck *et al.* grouped these two smells in Not Defined category.

- **Comments** In object oriented practice, code itself should tell its purpose. Using comments to describe code is often considered as a smell.

- **Incomplete Library** Partially implemented library create severe understandability issue for developers, hence it is considered as a code smell.

In this section, it is presented what code smells are and how those are classified. The next section describes the smell detection mechanisms proposed in previous researches.

2.3 Detection of Smells

As code smells are unavoidable problems in software development [3], those are needed to be identified and tracked. Many studies have concentrated on detection of code smells. The approaches to detect smells can be categorized in three groups. Those are - metric based approaches, correction based approaches and visual approaches. However, all three of these approaches have good and bad sides, and none of these are accepted as the single best way.

Metric based detection approach. Many researchers have proposed different metric based approaches to detect smells [28]. For example, Travassos *et al.* [23],

in 1999, proposed a manual smell detection technique where they calculated code metrics such as lines of code or variable count, and compared those against certain threshold values to identify instances of smells. This research was followed by few metric based heuristic techniques to fully automate smell detection [29, 30, 31]. In [32, 33], researchers introduced a set of rules for determining thresholds for smell detection, while Khomh *et al.* used Bayesian network to determine these threshold values [34]. However, non of these techniques are accepted as the standard for smell detection [35]. The most used detection based technique is Decor [36]. It allows user to adjust threshold values to find out anomalies that she wishes to detect.

Correction based detection approach. In the correction based approach, smells are identified based on available refactoring opportunities. For example, if there is an opportunity to extract a method from another method, that means there exists a long method which should be refactored. The advantage of this approach is, it identifies smells and the way to refactor those simultaneously. In JDeodorant, detection and refactoring of God Class [37], Long Method [37], Feature Envy [38] and Type Checking [39] smells are carried out using this technique. However, the problem with this technique is there are too many false positive results. For example, in this approach, even if a small manageable method is refactorable, it is marked as smelly, which is against the definition of code smells.

Visual based detection approach. As there exist significant amount of confusion about definition and detection of code smells, many researcher opted to leave the duty of smell detection on manual inspection. Simon *et al.* visually represented code metric to aid developers identifying design anomalies [24] while Dhambri *et al.* graphically presented bad design practices that can cause code smells [40]. As an extension to these works, Langelier [41] proposed a semi-automatic approach, where potential code smells and charecteristics of those are visual presented to the user to

identify bad smells [41]. Although these approaches are very helpful for identifying smells, these require manual inspection which is costly and very inconsistent in terms of accuracy.

In this section, the approaches of smell detection is discussed. The impact of code smell in software development is discussed in the next section.

2.4 Impact of Smells

Code smells have big influence on the evolution of design structrues [10]. Existence of smells increases software maintenance cost in terms of developers effort and time [16], and makes source code error prone [42]. Studies on impact of code smell proved that classes and methods containing code smell are more change prone and the change size in files containing code smells are usually larger than changes in smell-free files [25]. Therefore, it can be said that code smells have big influence in overall software quality and should be studied in detail.

Smelly code is difficult to maintain. According to the definition, code smells create numerous maintainability issues in a software such as decreased understandability [5] or decreased changeability [43]. Deligiannis *et al.* [10], [9] first studied the impact of code smells (such as Blob classes) on software development and maintenance activities, and found that smells influence the creation of badly structured code. Du Bois *et al.* [44], in 2006, performed a controlled experiment, where they decomposed God Classes into a number of small classes by performing refactoring and found that decomposed classes have a significantly higher understandability than the original God Classes. In 2013, Yamashita *et al.* observed a host of industrial java projects and the developers of those projects while they performed various maintenance tasks. Findings of this investigation show that, most of the maintenance problems are caused

by single or multiple code smells [45]. Studies, such as [46, 47, 48] inspected changeability of large commercial systems, and found that smelly classes are more resilient to change and thus, difficult to fix or enhance. Therefore, it can be seen from the literature that, code smells severely reduce maintainability of source code.

Smelly code is more fault-prone. As smelly code segments are less maintainable, the likelihood of bug initiation in these segments is very high [49]. In 2004, Vokavc *et al.* found that classes that follow good design patterns are less fault prone than others [42]. Zazworka *et al.* first investigated the relation between code smells and fault proneness of code, and found that God Classes are more defectprone than non-God Classes. Many other researches concentrated on the relation of code smells and fault proneness such as [49, 50, 51]. Results of these researches complement the findings of earlier researches [42, 20].

Smelly code is more change-prone. As smelly codes are more fault prone, developers have to fix those codes more frequently. Bieman *et al.* [52] analysed five systems to study change proneness of smelly code segments. They found that, classes with code smells have higher change frequency. To identify impact of different code smells, S. Olbrich [53] inspected change frequency and change size in Lucene and Xerces. They found that, files containing God Class and Shortgun Surgery smells are more change prone than files which do not contain these smells, and the average change size is also significantly higher in these files.

2.5 Relationships Between Smells

Complex relations can be identified between existing smells such as co-relation, co-occurrence, etc. [54, 16]. A brief discussion about researches which concentrated on code smell relations is presented in this section.

Co-relation of smells. To understand the co-relation between different code smells, Lozano *et al.* performed an empirical research on 95 versions of three open-source applications (Log4j, Jmol and JFreeChart) [11]. They inspected five smell relations (Plain Support, Mutual Support, Rejection, Common Refactoring, and Inclusion) from the seven defined by Pietrzak *et al.* [8] and four bad smells (God Class, Long Method, Feature Envy, and Type Checking). From this analysis they discovered correlations between God Class, Feature Envy and Long Method smells. Feature Envy and Long Method showed the strongest co-relation while Long Method-God Class, and Feature Envy-God Class showed mild cor-relation. These findings of this study are evidence of the co-existence of bad smells.

Co-occurrence of smells. To understand relations among code smells, and the frequency of those, Fontana *et al.* investigated 74 open-source system and found that a significant percentage of smell instances are related to other smell instances [1]. They found that 26% of God Classes use at least a Data Class, while 53% of Shotgun Surgeries and 70% of Dispersed Couplings are connected to at least one other smell. This observation confirms the theories that code smells tend to cluster together by architecturally connecting to other smells. They also inspected co-occurrence of code smells and found that Long Method has the largest share of co-occurrences (10%). Since Long Methods are high in size and complexity, if has a relatively higher chance of being affected by other smells. This study proves that code smells have a tendency to co-occur in architecturally connected code. However, in this study it is not addressed that how these relations between smells change with time. Therefore, there remains a scope for research for understanding the evolution of these co-occurred smells.

Impact of related smells. So far from the literature, it is seen that, code smells have complex relationship (co-relation and co-occurrence) patterns. Abbes *et al.* tried to find out how relations between smells effect the understandability of a system [2].

They gave developers a set of tasks to perform on smelly code, and analyzed their performance by using the NASA task load index, the times they spent performing the tasks, and the accuracy of the solutions. Results showed that the occurrence of one smell does not significantly decrease the understandability of source code. However, combination of two smells significantly decreases understandability.

From the literature, it is discovered that code smell co-occur in architecturally related code segments. It is also seen that a combination of co-occurred smells has worse manageability. Therefore, to have a better understanding of code smell, co-occurred code smells should be tracked and the evolution of those should be studied. In the next section, researches on evolution of code smells are discussed.

2.6 Evolution of Smells

Emergence of Code Smells is a recurring problem is software development which can not be solved permanently [3]. Smells reside in source code and evolve with time [7]. A large portion of recent studies on code smell concentrate on evolution process and pattern of smells.

Life cycle of smells. In 2010, Chatzigeorgiou *et al.* [7] studied the evolution of Long Method, Feature Envy, and State Checking throughout successive versions of two open-source systems to understand life cycle of code smells. They found that a significant percentage of smells are introduced during the addition of new methods to the code. If no intentional refactoring is performed, these smells reside in the code and evolve with it over time. While Peters *et al.* [3], in 2012, showed that even if smells are refactored, it is highly likely that those will recur in later versions. Therefore, it can be understood that emergence of code smell is a recurring problem in evolution of a software, and thus, it should be tracked and observed closely.

Change behavior. Using 9 releases of Azureus and 13 release of Eclipse, Khomh *et al.* [25] investigate the evolution process of code smells to understand the change behavior of smelly classes. They found empirical evidence that, in the evolution process, classes with smells tend to change more than classes without smells. In 2011, Zazworka studied evolution of individual smells and found that God Classes are changed more often [20]. This result of this study complements earlier findings which suggest - smelly code tends to change more in the evolution process [25].

Pattern of evolution. Olbrich *et al.* [53] tried to understand evolution pattern of smells by analysing the historical data of open source systems. After inspecting Lucene and Xerces for several years they concluded that -

- The total number of code smells increases steadily with time
- The relative number of components having code smells increases over time
- Components with smells is more likely to change
- Change size in smelly components is relative higher than non-smelly componants.

They also found that, Blob classes and Shotgun Surgery classes have a higher change frequency than other classes. This is the first study to present evolution patterns of code smells. However, this study did not inspected the evolution of code smell relations.

From the literature of code smell evolution, it can be seen that, code smells tend to reside in code, change frequently and increase in number with time. However, it is still unknown, how the evolution process of smells effect different relationships between smells.

2.7 Summary

Bad design practices that degrade the quality of a software are known as code smells [5]. There is no accepted standard for automatically detecting code smells, however there exists some widely used techniques [36, 38, 39, 37] to identify 22 re-known type of code smells [27]. These code smells significantly decrease maintainability [7] and changeability [25] of a software. Code smells existing in a software, change and increase with time [53], and often are related (co-related [8] or co-occurred [1]) to each other. However, it is not yet studied, in the evolution process of software, how the relationship between code smell change.

Chapter 3

Framework for Understanding the Evolution of Smell Clusters (FUESC)

A framework named as 'Framework for Understanding the Evolution of Smell Clusters (FUESC)' is developed and proposed in this chapter to extract evolution pattern of clustered smells. Using this framework, evolutionary behavior of smelly components can be detected, which will help to achieve a better insight of code smell evolution. Using 'FUESC', multiple consecutive versions of a software can be analyzed and clustering behavior from those can be extracted. This technique, at first, spots smell clusters for each version of a software one-by-one. After identifying smell clusters for all the versions, it observes how clustering behavior changed from one version to the next. Ultimately it examines the changes in clustering during the releases and extracts the patterns of evolution.

In this chapter, details of the proposed 'FUESC' framework is presented. At first, a brief overview is given to help the readers understand how this process works.

After that four steps of this technique, which are - smelly component detection, smell relation extraction, smell cluster identification and clustering behavior inspection are described. Finally, a summery of the whole chapter is presented at the end of the chapter.

3.1 Overview

Code smells residing in software source code are often architecturally inter-connected. These connected smells create clusters of bad smells which participates in the evolution process of a software. In this study, the key focus is to understand how relationships between different code smells evolve with time by observing the behaviors of code smell clusters. Firstly, smelly clusters for different releases of a software project are identified and their properties are extracted. By comparing these properties, evolution pattern for smell clusters are discovered. A greater understanding of code smell relationships is achieved by examining the evolution pattern of these smell clusters. An example situation is presented in Figure 3.1.

In Figure 3.1 (Top-Left) , an example set of classes is shown along with its methods. Here, the square boxes represent classes and the oval-shaped ones represent methods. The relationships between a class and a method is shown as an undirected edge. In Figure 3.1 (Top-Right), detected smelly components are shown using gray boxes. Then, in Figure 3.1 (Bottom-Left) the relationships between the components are depicted. Different relationships are illustrated as different types of edges such as directed edges represent a call from a method to another method, dual edges represent relationship between two classes. Finally, Figure 3.1 (Bottom-Right) shows the extracted smelly component clusters.

The main work of this technique is divided into four major steps as shown in Figure

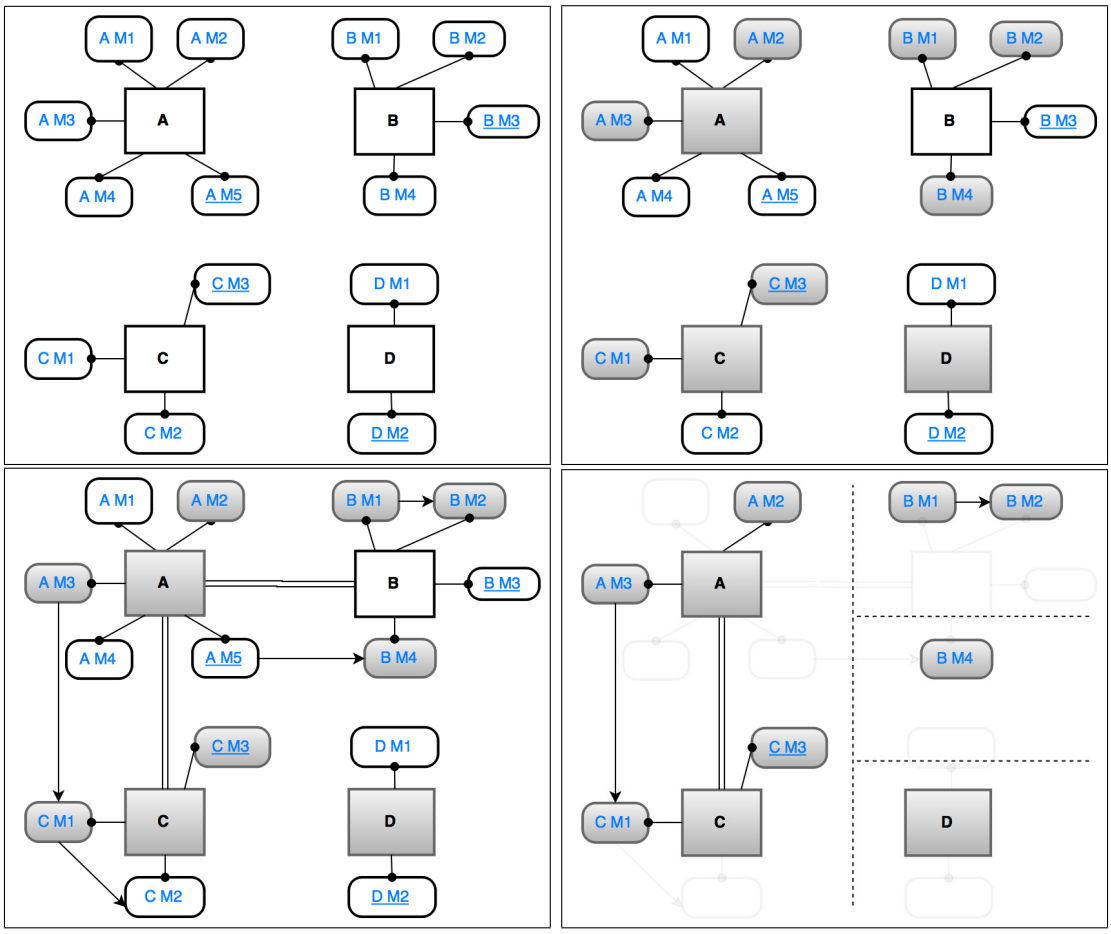


Figure 3.1: Smelly Cluster Detection Process

3.2. The concern of the first three steps is to spot out and profile smell clusters. After obtaining the clustering property (such as cluster count, cluster sizes, etc.) for all versions of a software, the final step is initiated. The focus of this step is to identify evolution patterns of smell clusters by comparing previously extracted clustering data for all versions of a software. These four major works performed in this process are given below -

- **Smelly Component Detection** To detect smell clusters, first, all the smelly components must be detected from source code. The purpose of this step is to detect these smelly components. To do so, all instances of code smell existing are located. Classes that contain one or more smells are noted as smelly classes. Similarly, methods containing one or more smells are identified as smelly meth-

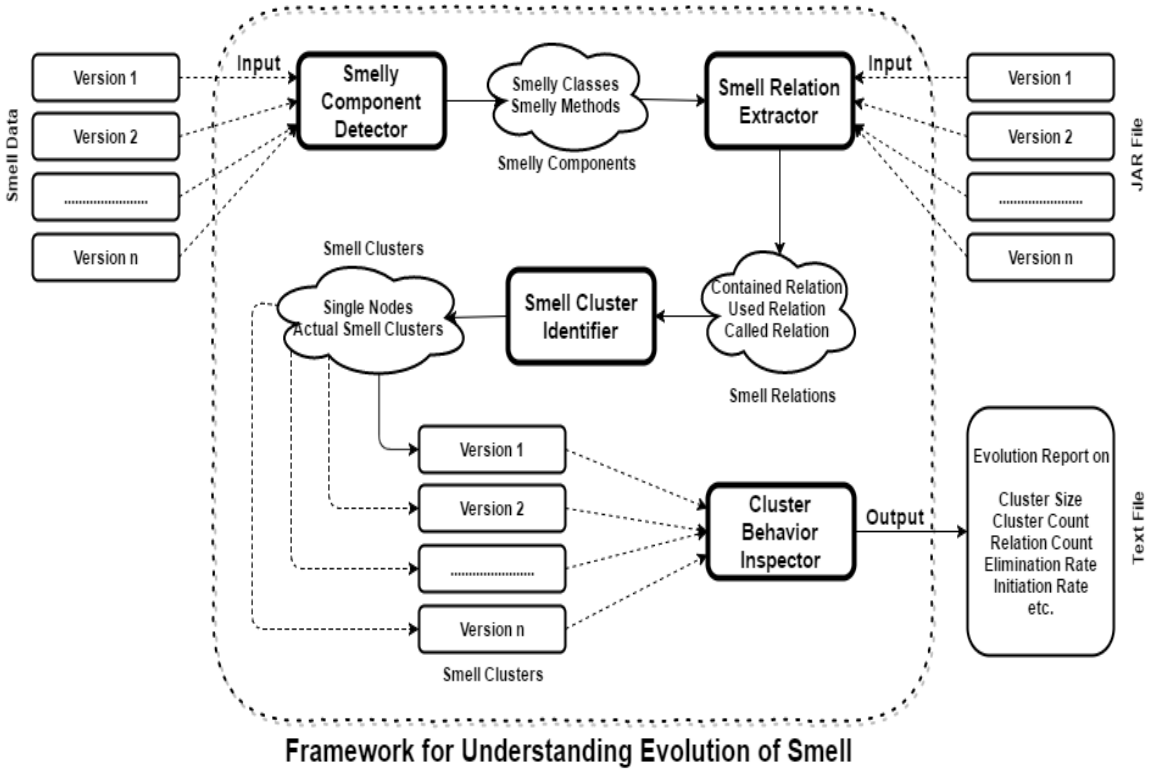


Figure 3.2: Overview of 'FUESC'

ods. Ultimately, a list of smelly components (smelly methods and smelly classes) [6] is created.

- **Smell Relation Extraction** To Detect smelly clusters from from smelly components it is required to extract relationships between different smells. Therefore, in this step, the architectural relationships between different smelly components (class-class, class-method, method-method relationships) are collected from source code.

- **Smell Cluster Identification** A graph is generated using data from previous two steps. Every smelly component is inserted in the graph as a vertex and every relation between these components is added as an edge between those. After producing this graph, it is searched using graph-search algorithm (BFS) to find connected clusters of bad smells. The properties of discovered smell

clusters (such as cluster count, cluster sizes, etc.) are profiled.

- **Clustering Behavior Inspection** In this step, clustering behaviors recorded for different versions of a software are analyzed to identify evolution patterns.

So far, the an overview of the process is presented along with an example case. In the next four sections, four key steps of the 'FUESC' framework are described one-by-one.

3.2 Smelly Component Detection

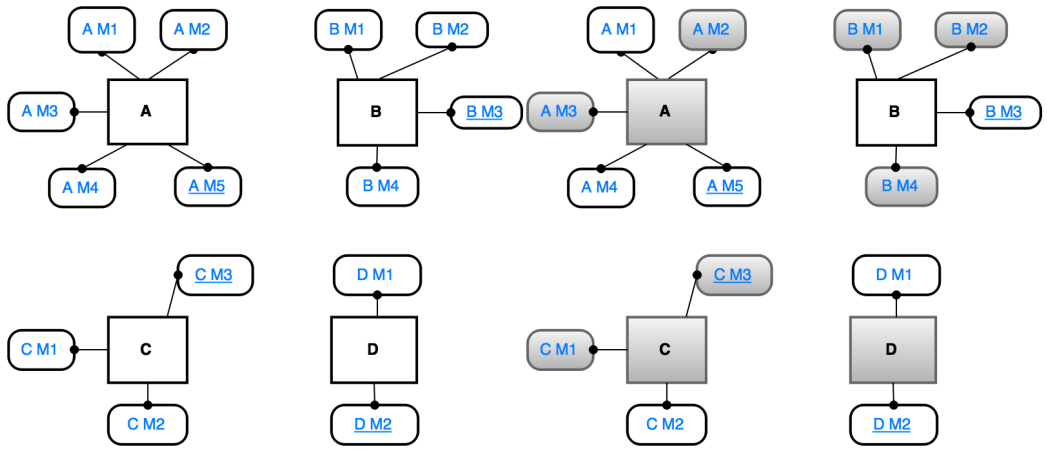


Figure 3.3: Smell Detection

In this step smelly code components (such as smelly classes and smelly methods) are detected from source-code (Figure [?]). Later in this study, these detected smelly components are used to identify clusters of bad smells. In order to do that, the challenge is to select which smells to target and how those should be detected. In this section, this decisions are discussed in detail. Before going to smell detection process, few challenges regarding this process needs to be discussed.

3.2.1 Decisions Regarding Smell Detection

The key problem in smell detection is, there is no accepted standard for detecting code smells [35]. For example, when a class gets extremely large in size that it becomes very difficult for developers to understand, maintain or update, it is called a god class. However, the exact number of lines required for a class to be a god class is not defined in literature. Fowler [5] described that a class is a god class when it contains 750 or more lines of code. Here, the question is if a class is 749 lines long instead of 750, is there any significant difference in it's maintainability? Different researchers have defined different threshold values for detecting code smells. However, none of them are considered as standard. That is why the challenge is to decide which standard to follow to detect smells.

Fowler also stated that [5], the best way to detect code smells is using human intuition. However, for a large project, detecting code smells manually is a highly time consuming and costly approach. Depending on the expertise and experience of developer, the output of manual smell detection process might be extremely inconsistent.

Although the standards of smell detection is not established yet, there are few tools which are widely used for smell detection. For example, JSNose [55], DECOR [36], JDeodorant [39], etc. Any of these tool might be used for smell detection, as the key concern of this study is not to detect smells, but to analyze relationships between those. That is why the standards of detection do not carry much significance in this case.

3.2.2 Smells to Detect

The proposed framework can identify evolution pattern for any smell if smell detection data is provided to it. The selection of smells depend on the tool which is used with

this framework. The tool, JDeodorant [39], which is used for this research, currently supports detection of four smells - God Class, Long Method, Feature Envy and Type Checking. That it why these four smells where taken into consideration for this framework.

- **God Class (GC)** A 'God Class' is a class that has grown extremely large is size, and has become very difficult for a developer to understand, maintain or update. It controls too many other classes in the system and has grown beyond all logic to become 'The Class That Does Everything'. A class is usually considered by judging the number of code lines it contains or by the amount of methods it holds.

- **Long Method (LM)** A 'Long Method' is a function or procedure that does not focus on only one task, rather it performs a series of tasks and works as the brain of a class. Long methods are detected by analyzing its scope of work and line count.

- **Feature Envy (FE)** A class is considered to have 'Feature Envy' smell when it uses methods of another class excessively. This incident indicates that the structure of the feature envy class or the excessively used method is wrongly designed. 'Feature Envy' classes are generally judged by observing its interaction with methods of other classes.

- **Type Checking (TC)** When the type of an object is explicitly checked in software code it is considered as a 'Type Checking' smell. This smell indicates that key concept of object oriented programming such as encapsulation, polymorphism and inheritance were not properly followed in the program. These smells are simply identified by finding statements containing 'typeOf', 'sizeOf', 'instanceOf', etc keywords.

3.2.3 Smell Detector

JDeodorant [39] is used for detecting instances of code smells in software's code. This tool follows a correction based approach to detect of four smells - A) God Class (GC) B) Long Method (LM) C) Feature Envy (FE) D) Type Checking (TC). In correction based approach, code smells are spotted by identifying refactoring opportunities. In simple words, it means if there is a scope to improve a solution, there exists problems in it.

Using JDeodorant, all instances of these four types of smells are detected in each version of software. JDeodorant produces one smell file for each type of smell, which contains a list of all occurrences of that type. As God Class and Feature Envy are class level smells, smell files for God Class and Feature Envy contain list of smelly classes. Similarly, as Long Method and Type Checking are method level smells, smell files for these smells contain list of smelly methods. This files are parsed by a smell parser to create a single unified list of smelly components, which is used later in 'FUESC'.

3.2.4 Parsing Smell Data

After the detection of code smells in a software, a list of smelly methods and smelly classes is generated. These lists are combined to generate a general list of smelly classes and smelly methods.

First two empty lists of smelly classes and smelly methods is created. For each smell file, the list of smells is iterated. For each smell, first it is checked whether it is a method level or a class level smell. If it is a method level smell, and is not already in the smelly method list, it is added to this list. Same process is followed for class level smell as shown in Algorithm 1 (line 17-21). This whole process is followed for each smell type. When all the smelly methods and smelly classes are listed, a unified

list of smelly components is found.

Algorithm 1 Parsing Smell Data

```
1: files: Smell Detector Output Files
2: procedure ParseSmellData(files)
3:     smellyMethods ← empty
4:     smellyClasses ← empty
5:     size ← files[size]
6:     for i ← 0 to size do
7:         smellFile ← files[i]
8:         smells ← smellFile.getSmells()
9:         smellCount ← smells[size]
10:        for j ← 0 to smellCount do
11:            smell ← smells[j]
12:            if smell[type] = methodLevel then
13:                if smell is not in smellyMethods then
14:                    smellyMethods.add(smell)
15:                end if
16:            end if
17:            if smell[type] = classLevel then
18:                if smell is not in smellyClasses then
19:                    smellyClasses.add(smell)
20:                end if
21:            end if
22:        end for
23:    end for
24:    smellyComponents := smellyMethods + smellyClasses
25:    return smellyComponents
26: end procedure
```

After parsing, a list of smelly components is found, which is used as input in the 'Smell Cluster Detector' to identify smell clusters. In the next section, the relation extraction process is described.

3.3 Smell Relation Extraction

To identify smell clusters, it is needed to extract the relationships between different smells. This step of the process is concerned with smell relationship extraction

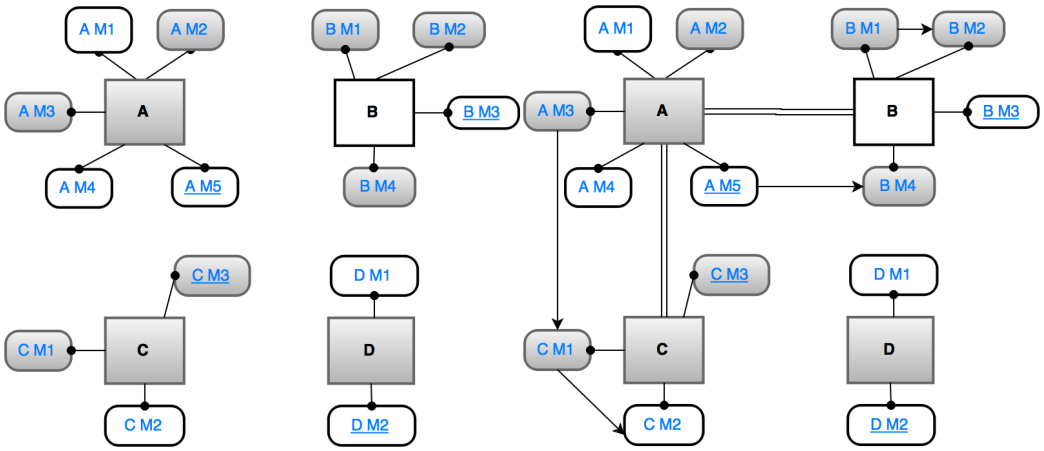


Figure 3.4: Smelly Components

using the list of smelly components as shown in Figure 3.4. From the background study, it is seen that Code smells are more likely to stay inter-connected in software rather than remaining as a single instance. These type of relationships create architectural connection between smells. Different architectural connections (class-class, class-method or method-method) can be observed between smelly components of a software. For instance, it is a highly common scenario that a god classes contain long methods, or god classes are envious toward methods of other classes. Different types of architectural relationships between smells are given below.

3.3.1 Relationship Types

As defined in [1], there exists three types of relationships between smelly componants, which are -

- Contained Relation: Relation between a smelly class, and a smelly methods contained inside it, is called contained relationship, for example, a Long Method inside a God Class.
- Used Relation: When a smelly class is used by another smelly class, those two have used relationship, for example, a god class using a data class.

- Called Relation: When a smelly method is called by another smelly method, those to have called relationship, for example, a long method calling another long method.

Relationships between smells are identified in two steps. The first step is to analyze the software source code, and extract its architecture. From the extracted architecture of the software 'Contained Relations' can be identified. In the second step, call graph is generated which is used to identify 'Used Relations' and 'Called Relations' between smells.

3.3.2 Architecture Extraction

The purpose of architecture extraction is to find 'Contained Relations' between smells. At first, an Abstract Syntax Tree (AST) is generated from source code of a software. From the AST, for every smelly classe, contained methods are identified. For all the identified methods which are smelly, a relation pair is created between the class and the method as shown in Algorithm 2 (line 16). This relation is added to the list of contained relation list. Finally a list of contained relations is found.

3.3.3 Call Graph Extraction

The purpose of this step is to find 'Used Relations' and 'Called Relations' between smells. At first a call graph, a directed graph that shows the calling relationship between different subroutines of a program, is generated from source code. By analyzing the call graph it is found that which method calls which method and which class is used by which class. This information is used to identify 'Used Relations' and 'Called Relations' between smells. In this step of the work, call graph for the targeted system is generated from the binary code and the following two types of inter-smell

Algorithm 2 Contained Relation Extraction Algorithm

```
1: source: Software source code
2: smells: List of smelly components
3: procedure GetContainedSmellRelations(source, smells)
4:     containedRelations ← empty
5:     AST ← ASTParser(source)
6:     classes ←AST.getAllClasses()
7:     cCount ←classes.size
8:     for i ← 0 to cCount do
9:         class ← classes[i]
10:        if class is in smells then
11:            methods ←AST.getContainedMethods(class)
12:            mCount ←methods.size
13:            for j ← 0 to mCount do
14:                method ←methods[j]
15:                if method is in smells then
16:                    relation ← r(class to method)
17:                    containedRelations.add(relation)
18:                end if
19:            end for
20:        end if
21:    end for
22:    return containedRelations
23: end procedure
```

relation is identified - Used Relation and Called Relation. The whole process is given in Algorithm **??**.

Algorithm 3 Used Relation Extraction

```
1: binary: Binary code generated from source code
2: smells: List of smelly components
3: procedure GetUsedSmellRelations(binary, smells)
4:     usedRelations ← empty
5:     classes ← binary.getAllClasses()
6:     cCount ← classes.size
7:     for i ← 0 to cCount do
8:         class ← classes[i]
9:         if class is in smells then
10:            usedClasses ← binary.getUsedClasses(class)
11:            ucCount ← usedClasses.size
12:            for j ← 0 to ucCount do
13:                usedClass ← usedClasses[j]
14:                if usedClass is in smells then
15:                    relation ← r(classtousedClass)
16:                    usedRelations.add(relation)
17:                end if
18:            end for
19:        end if
20:    end for
21:    return usedRelations
22: end procedure
```

By combining the software architecture derived from source code, software call graph derived from binary code with software smell data generated by third party smell detector, the relationship between all smells are extracted.

3.4 Smell Cluster Identification

In the Cluster Identification step, smell clusters are identified by creating a graph using smell data and applying graph-search algorithm in it. Every smelly components are considered as nodes and every relationships between two components are consid-

Algorithm 4 Called Relation Extraction

```
1: binary: Binary code generated from source code
2: smells: List of smelly components
3: procedure GetCalledSmellRelations(binary, smells)
4:     calledRelations ← empty
5:     classes ←binary.getAllClasses()
6:     cCount ←classes.size
7:     for i ← 0 to cCount do
8:         class ← classes[i]
9:         methods ←binary.getContainedMethods(class)
10:        mCount ←methods.size
11:        for j ← 0 to mCount do
12:            method ←methods[j]
13:            if method is in smells then
14:                calledMethods ←binary.getCalledMethods(method)
15:                cmCount ←calledMethods.size
16:                for k ← 0 to cmCount do
17:                    calledMethod ←calledMethods[k]
18:                    if calledMethod is in smells then
19:                        relation ← r(methodtocalledMethod)
20:                        calledRelations.add(relation)
21:                    end if
22:                end for
23:            end if
24:        end for
25:    end for
26:    return calledRelations
27: end procedure
```

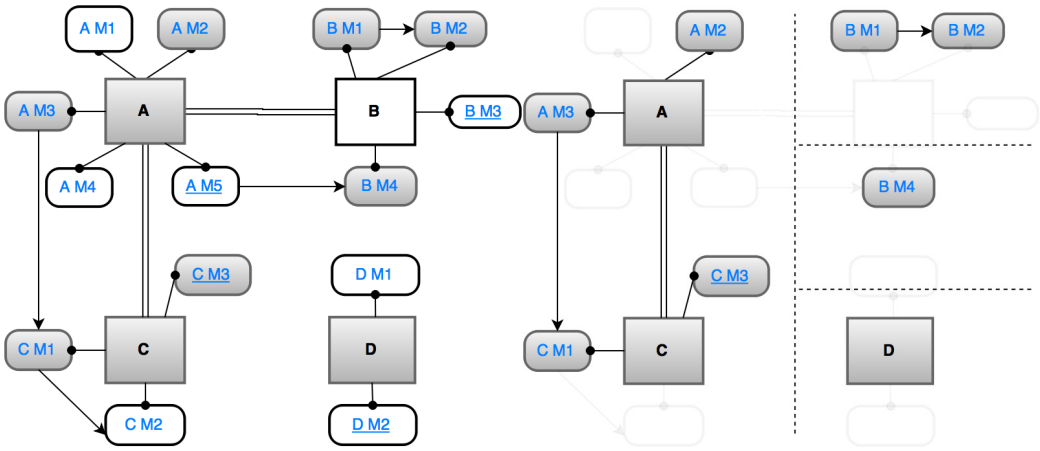


Figure 3.5: Relation between Components

ered as edges to generate a graph. In this graph, different types of edges are used to show class-class, class-method, method-method relationships as shown in Figure 3.5.

After the graph generation, clusters are identified using graph search algorithms such as Breath-First-Search (BFS), Depth-First-Search (DFS). Finally, various characteristics of smells such as clustering behavior, connectivity, change of cluster size, are extracted by analysing the clusters.

Using this process, smell clusters are identified for various versions of software. Then these clusters are analyzed to identify the evolution of the code smell relationships.

3.4.1 Graph Generation

After detection of code smells and their architectural connection, a graph is generated for cluster detection. Every smelly component is inserted in the graph as a node. After that, an edge is drawn between every pair of architecturally connected smells. This process stops when every connection between smells is inserted in the graph. Multiple connection between two nodes is also represented by single edge as shown in Algorithm 5 (line 14-21). After this process, a graph of smelly components for a

specific version of a software is found.

Algorithm 5 Graph Generation Algorithm

```
1: smells: List of smelly components
2: relations: List of relations between smelly components
3: procedure GenerateGraph(smells, relations)
4:     empty graph G = (V, E)
5:     list of vertexes V ← empty
6:     list of edges E ← empty
7:     sCount ←smells.size
8:     for i ← 0 to sCount do
9:         smell ← smells[i]
10:        V.add(smell)
11:    end for
12:    rCount ←relations.size
13:    for j ← 0 to rCount do
14:        relation from u to v, r(u, v) ←relations[j]
15:        if u is in V then
16:            if v is in V then
17:                if r(u, v) is not in E then
18:                    E.add(r(u, v))
19:                end if
20:            end if
21:        end if
22:    end for
23:    return G
24: end procedure
```

Here, smells is the list of smelly components and relations is the architectural relation of the whole system.

3.4.2 Cluster Detection

When the graph is ready, graph search algorithm is used to identify connected clusters of code smells. Breath-First-Search (BFS) is used here for searching graph. Instead of BFS, DFS termed as Depth-First-Search could also be used, which would produce the same results. For each node of the graph, BFS is performed to gather all the nodes which are reachable from it. All the reachable nodes are removed from the original

graph. This set of reachable nodes is a cluster of smells. This cluster is added to the smell cluster list as shown in Algorithm 6.. This process is repeated until the original graph is empty. By that time the list of smell clusters is full with all the cluster.

3.5 Clustering Behavior Inspection

Finally, after detecting all the smell clusters, the clustering behavior for that specific version is saved to compare with other versions of the software. The following properties are calculated for this research.

- **GC Count** Total number of God Class smells
- **LM Count** Total number of Long Method smells
- **FE Count** Total number of Feature Envy smells
- **TC Count** Total number of Type Checking smells
- **Smelly Class Count** Total number of smelly classes
- **Smelly Method Count** Total number of smelly methods
- **Classes Containing Smells** Total number of classes that either is smelly or contains at least one smelly method
- **Used Relation Count** Total number of relations between two smelly classes
- **Contained Relation Count** Total number of relations between a smelly class and a smelly method
- **Called Relation Count** Total number relations between two smelly methods
- **Vertex Count** Total number of smelly nodes in the graph

Algorithm 6 Cluster Detection Algorithm

```
1: smells: List of smelly components
2: relations: List of relations between smelly components
3: procedure DetectClusters(smells, relations)
4:     list of vertexes V ← smells
5:     list of edges E ← relations
6:     clusters = list of cluster c(nodes, edges)
7:     clusters ← empty
8:     vCount ←V.size
9:     for i ← 0 to vCount do
10:        smell ← smells[i]
11:        connectedNodes ← getConnectedNodes(smell, relations)
12:        clusters.add(connectedNodes)
13:        V.remove(connectedNodes)
14:    end for
15:    return clusters
16: end procedure
17: procedure GetConnectedNodes(smells, relations)
18:    list of vertexes V ← smells
19:    list of edges E ← relations
20:    cluster with smells nodes and relations edges
21:    cluster ← empty
22:    vCount ←V.size
23:    for i ← 0 to vCount do
24:        smell ← smells[i]
25:        rCount ←relations.size
26:        nodes.add(u)
27:        for j ← 0 to rCount do
28:            relation from u to v, r(u, v) ←relations[j]
29:            if u is in V then
30:                if v is in V then
31:                    if r(u, v) is in E then
32:                        nodes.add(v)
33:                        edges.add(r(u, v))
34:                    end if
35:                end if
36:            end if
37:        end for
38:    end for
39:    return cluster = cluster(nodes, edges)
40: end procedure
```

- **Cluster Count** Total number of smell clusters in the graph
- **Cluster Sizes** List of cluster sizes existing in the graph
- **Largest Cluster Size** Size of the largest cluster
- **Single Node Count** Total number of Single-Node clusters in the graph
- **Nodes in Small Clusters** Vertex Count - Largest Cluster Size - Single Node Count
- **List of Clusters** A list of all clusters with all nodes in those

3.6 Summary

A framework 'FUESC' is proposed in this chapter that analyses software source code to identify evolution patterns of smell clusters. The techniques used for this process are illustrated in this chapter along with the reasons why those were selected. Four key sub-tasks is performed in this approach which are - 1) Smelly Component Detection, 2) Smell Relation Extraction, 3) Smell Cluster Identification and finally 4) Clustering Behavior Inspection. Task-1, task-2 and tast-3 extracts smell clusters from source code step-by-step. Task-4 consists of manual inspection of the results obtained by previous tasks. In the next chapter, few open-source software will be examined using this methodology to understand the evolution pattern of code smells residing in those.

Chapter 4

Smell Cluster Inspection Tool (SCIT)

Smell Cluster Inspection Tool (SCIT) is developed to find and inspect clusters of code smells that exist in a java project. This tool implements the FUESC framework that is proposed in Chapter 3. The task of smell cluster detection and pattern discovery is performed by four core units of this tool which are - smelly component detector, smell relation extractor, smell relation identifier and cluster behavior inspector. As an input this tool takes lists of instances of code smells and a jar file of the project and generates a 'clustering report' text file as an output. This output contains all the clustering behavior (such as evolution pattern of cluster sizes, smell initiation and elimination rates in clusters etc.) calculated by it. Using this tool, a case study is performed on 25 versions of three open-source java projects to understand evolution of smell clusters. The tool discovered that, number of smelly component and relationship between those increases as a software grows old. It also found that as time passes, new clusters of smells are introduced in the system and sizes of those increase steadily.

In the previous chapter, a framework called FUESC is proposed for understanding

clustering behavior of code smells. In this chapter, a tool SCIT, which is developed based on that framework will be discussed. In the following sections the architecture, input output and demonstration of SCIT is described. After that, detail about the performed case study is presented where the data sets, experiment setup, and results are discussed.

4.1 Smell Cluster Inspection Tool

SCIT is a tool that analyzes different versions of a software to extract change behavior of smell clusters. By examining this change behavior a deeper insight about the evolution of code smells can be achieved. SCIT is implemented in JAVA and currently operates on JAVA source code. It analyzes the smell data for different versions and creates a report after analyzing those. This report contains information about cluster sizes, different cluster's evolution pattern etc.

In the following subsections, the architecture of SCIT is described followed by a discussion about its input and output. After that a demonstration of the tool is given in a seperate subsection.

4.1.1 Architecture of SCIT

SCIT is a tool combining four different units - smelly component detector, smell relation extractor, smell relation identifier and cluster behavior inspector. The tasks performed by these units are described below.

- **Smelly Component Detector:** The smelly component detector detects individual smelly methods and smelly classes by analyzing smell data from third party smell detectors and generates a unified list of smelly components. There is a parser module inside this unit which parses third party data to create on

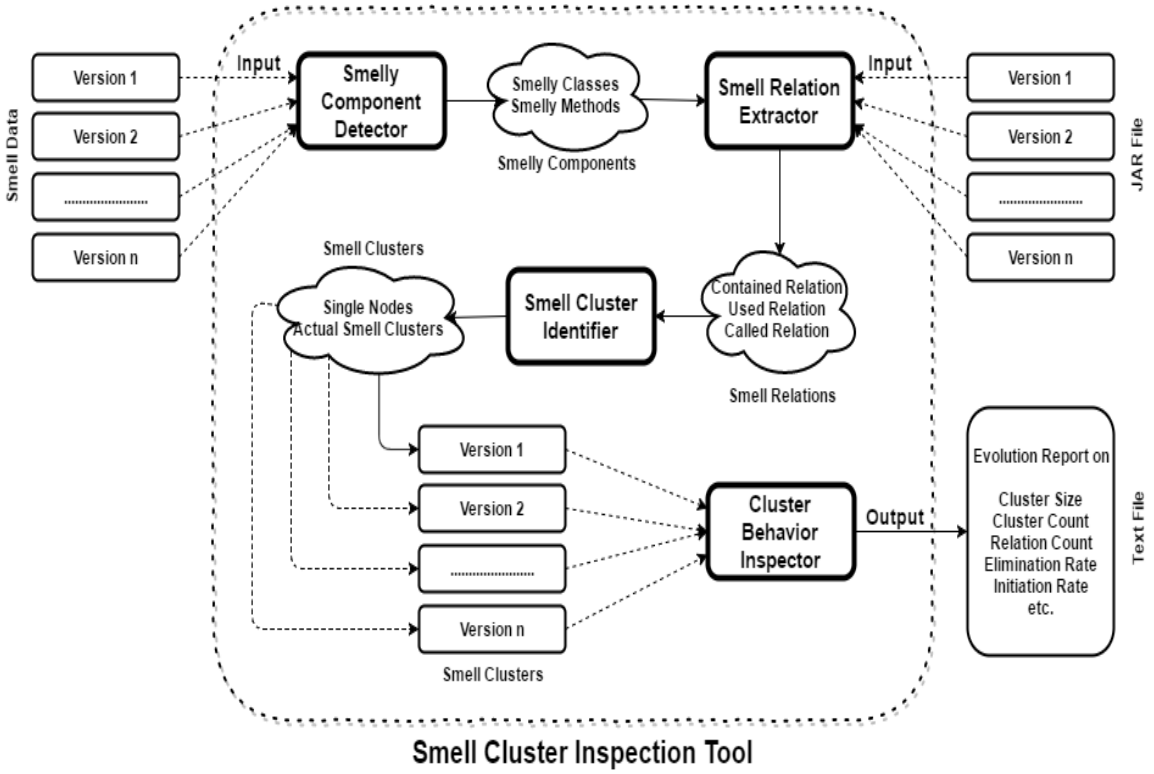


Figure 4.1: Architercture of SCIT

memory list of smells. For this study, a parser for JDeodorant data is built which parses four output files of JDeodorant which are lists of God Classes, Feature Envies, Long Methods and Type Checkings, and creates a list of smelly components. This list is used in the Smell Relation Extractor unit and Smell Cluster Identifier unit. For this study, only JDeodorant parser is created. To use data from other smell detectors (such as Ptdej, or DECOR), another new parser module can be plugged inside this unit.

- **Smell Relation Extractor:** This element of this toolkit has two duties, those are, extracting architecture from bytecode to find 'contained relation' between code smells, and finding 'used relation' and 'called relation' by analyzing call graph. The final output of this unit is a list of pairs of smelly components denoting the architectural connections between smells.

- **Smell Cluster Identifier:** This unit is concerned with graph generations from smell data and architecture data. It finds clusters of code smells in these graphs and profiles those as a set of smelly components.

- **Clustering Behavior Inspector:** The final part of this toolkit is a cluster analyzer. This analyzer compares clustering behavior (such as cluster growth, size, new smell initiation and elimination rate etc.) of different releases to identify pattern of evolution. This report is printed in a text file as the final output of the tool.

4.1.2 Input and Output of SCIT

SCIT takes lists of code smells and a jar file as input for each version of a software and generates a cluster evolution report text file as output. Instead of taking these files one by one it takes a path to all files and searches that path to find versions that can be analyzed. If it can find four list of smells (God class, Feature Envy, Long Method and Type Checking) and a jar file for a version, it considers the version as analyzable. SCIT shows the user the list of analyzable versions from which user can choose the versions that he wish to inspect shown in Figure 4.2.

SCIT produces one analysis file for each version of the project in the given path. This file contains clustering report on that specific version only. One cluster evolution report is generated combining the data of all versions and saved as text file in the given path. This file can be opened in CSV editor for a better presentation of the analysis.

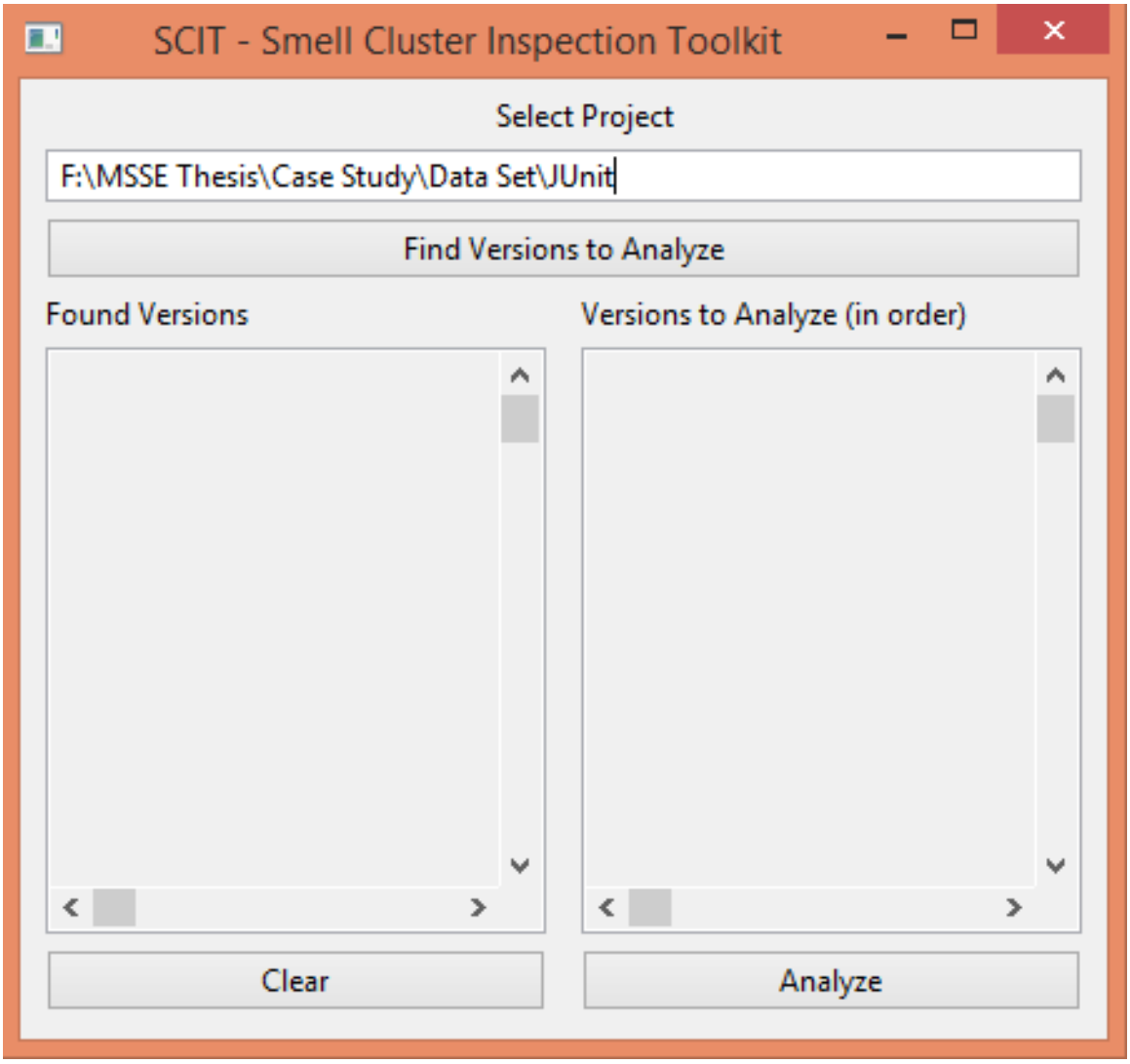


Figure 4.2: User Interface of SCIT

4.1.3 Demonstration of SCIT

In Figure 4.2, the user interface of SCIT is presented. The UI consists of one input field and three buttons. The input field takes a path to all files. At first the user have to paste a path to all files and press the button 'Find Versions to Analyze'. In the 'Found Versions' box in UI (Figure 4.2), a list of found analyzable versions is shown. Now, the user have to select all the versions he wants to inspect as shown in Figure 4.3. Here, the order of selection is very important because SCIT cannot automatically

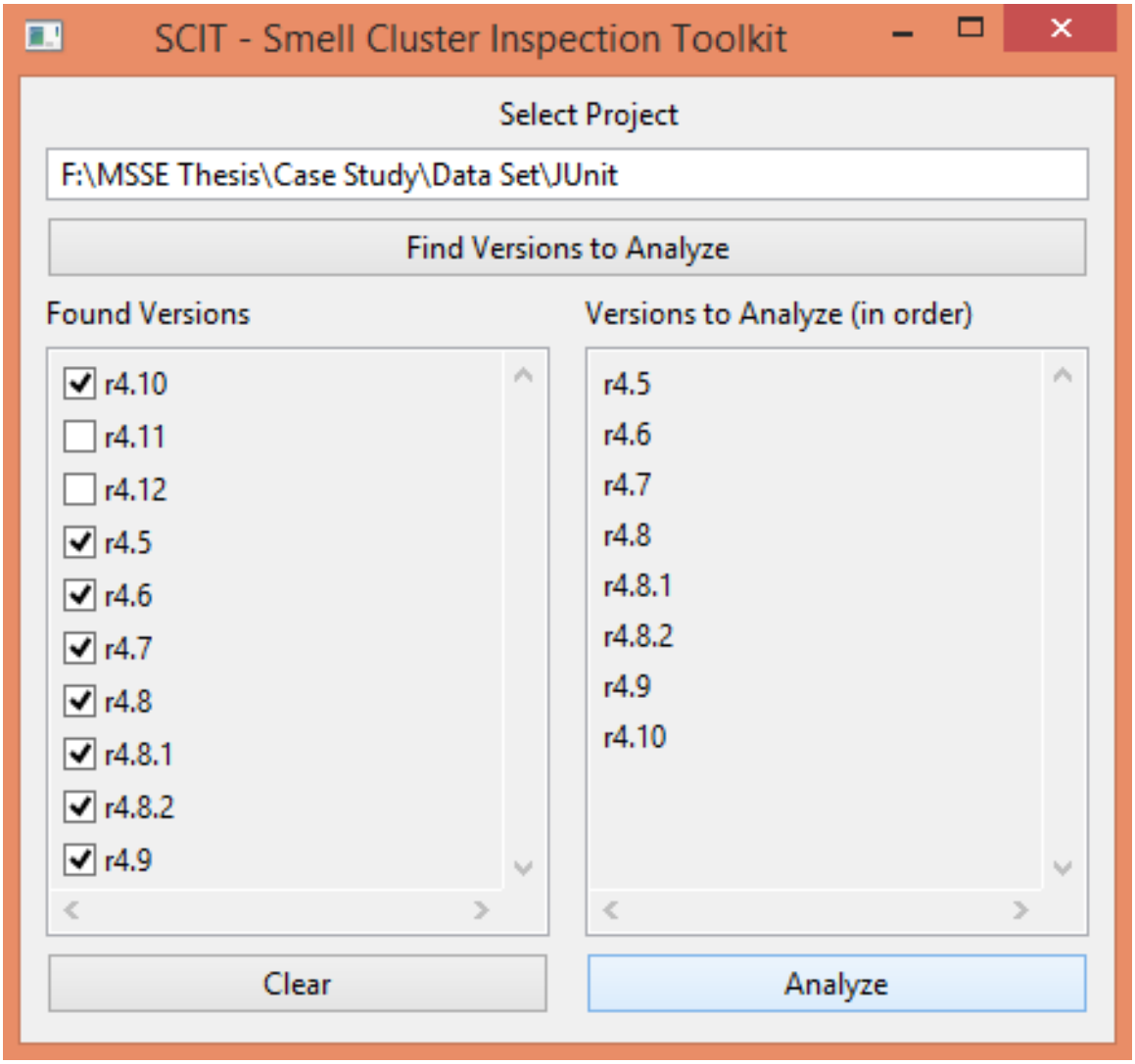


Figure 4.3: Demo: SCIT

understand which version was released after which. According to the user's selection order, SCIT shows an ordered list of versions in right hand side panel. If the user wants to re-select versions, she can press the clear button to start the process from beginning. If the ordering is correct, the user can press the 'Analyze' button to start the analysis. When the analysis is finished, the output file is saved in the same path that was given previously as input.

4.2 Dataset

The analysis was performed on 3 popular open source java libraries - JUnit [56], Apache Commons-lang [57] and mockito [58]. The selection of this projects were based on popularity and compilability of this libraries. The projects that were analyzed are described below -

- **JUnit:** JUnit is an open-source unit testing library in java. It is the most popular java library among open-source github repositories. According to a research [59] on github projects, more than 64% of all open-source java projects use JUnit for automated unit tests. The JUnit development started in 2004. Since then, 10 stable versions (from 4.5 to 4.12) of the library have been released. The fact that, JUnit has a significantly long history of development, makes it a good choice for this study.

- **Apache Common-lang:** Apache Commons-lang is an open-source utility library developed for java projects. It provides a host of utilities to support java-lang, for example, basic numeric and string manipulation methods. It is a highly popular library in java development. Over 12% of all open-source java projects in github use this utility library. The first release (version 1) of this library came in 2002. since then 4 other stable releases (version 3.0 , 3.1, 3.2, 3.3, 3.4 and 3.5) of it have been published. All the releases of the library are easily compilable and the project structure did not change much between different versions. These factors make apache Commons-lang a good selection for this study.

- **Mockito:** Mockito is the most popular java mocking library for writing clean and easily readable test codes. It is used by more than 30,000 open source java projects hosted in github. According to [59], 10.72% of git-hub open-source

projects use mockito library. The library was first written in 2008. Since then 96 releases of it was released. 14 of these versions were published as stable releases. 10 consecutive releases of this project from 1.5 to 1.9.0, which were similar in project structure, are studied in this investigation.

4.3 Environmental Setup

The case study was performed in the following setup -

- **Machine:** Intel core i7 second generation processor with clock speed of 2.93 GHz, 4GB of RAM.
- **Operating System:** Microsoft Windows 8 64bit operating system.
- **Java Virtual Machine:** Java runtime environment 7.
- **Smell Detector:** JDeodorant 5.0.42
- **Bytecode Generator:** Bytecode generated by Eclipes, Version: Mars.1 Release (4.5.1)
- **Source Code:** source code of different versions of JUnit can be found in Github under junit-team/junit4 [60]. Under mockit0/mockito source codes for Mockito can found in Github [61]. Commons-lang is hosted in Github under apache/Commons-lang [62].
- **SCIT:** The source code of the developed tool 'Smell Cluster Inspection Toolkit' is hosted in Dropbox [63].
- **Input Data:** JDeodorant's output is used as one of the inputs in the case study. These data can be found in the replication package [63].

4.4 Results of Case Study

As the tool performs the analysis in four different components, the results come in four stages. At the first stage, the output is a list of smelly components (smelly methods and smelly classes). Number of relations (such as contained, used and called relations) between smelly components is found in the second stage. At the third stage of result collection, a list of code smell clusters is gathered. In the fourth and final stage, evolutionary behaviors of smell cluster are collected as final output.

In the following four sections, the raw results for each of these four stages will be presented one by one. In the next chapter, it will be discussed how these results answer the research questions of this study.

Figure 4.4: Smelly Componants in JUnit

4.4.1 Results: Smelly Component Detection

In the first phase of the case study, smelly method list and smelly class list for JUnit, Mockito and Commons-lang is collected. To justify the validity of the dataset, it is important to check whether the smells exhibit the behaviors previously stated by other researchers. Alexander Chatzigeorgiou et al showed that, number of code smells in a software increases steadily with time [7]. Collected smelly clusters are inspected to see whether these follow this theory of code smell evolution and it is found that all three of the subjects exhibit this behavior.

As shown in Figure 4.4, the number of smelly methods residing in JUnit increased from 52 in the first release, release-4.5, to 74 in the final release, release-4.12. During these ten releases, smelly class count reached 55 from the initial value of 32. The overall increase of smelly components is 45 at an increase rate of 5 components per release which is 5% of growth per release.

As shown in Figure 4.5, the number of smelly methods residing in Mockito increased from 16 in the first release, release-1.5, to 59 in the final release, release-1.9.0. During these ten releases, smelly class count reached 30 from the initial value of 9. The overall increase of smelly components is 64 at an increase rate of 7.11 components per release which is 15% of growth per release.

As shown in Figure 4.6, the number of smelly methods residing in Commons-lang increased from 34 in the first release, release-3.0, to 104 in the final release, release-3.4. During these five releases, smelly class count reached 23 from the initial value of 18. The overall increase of smelly components is 75 at an increase rate of 18.75 components per release which is 25% of growth per release.

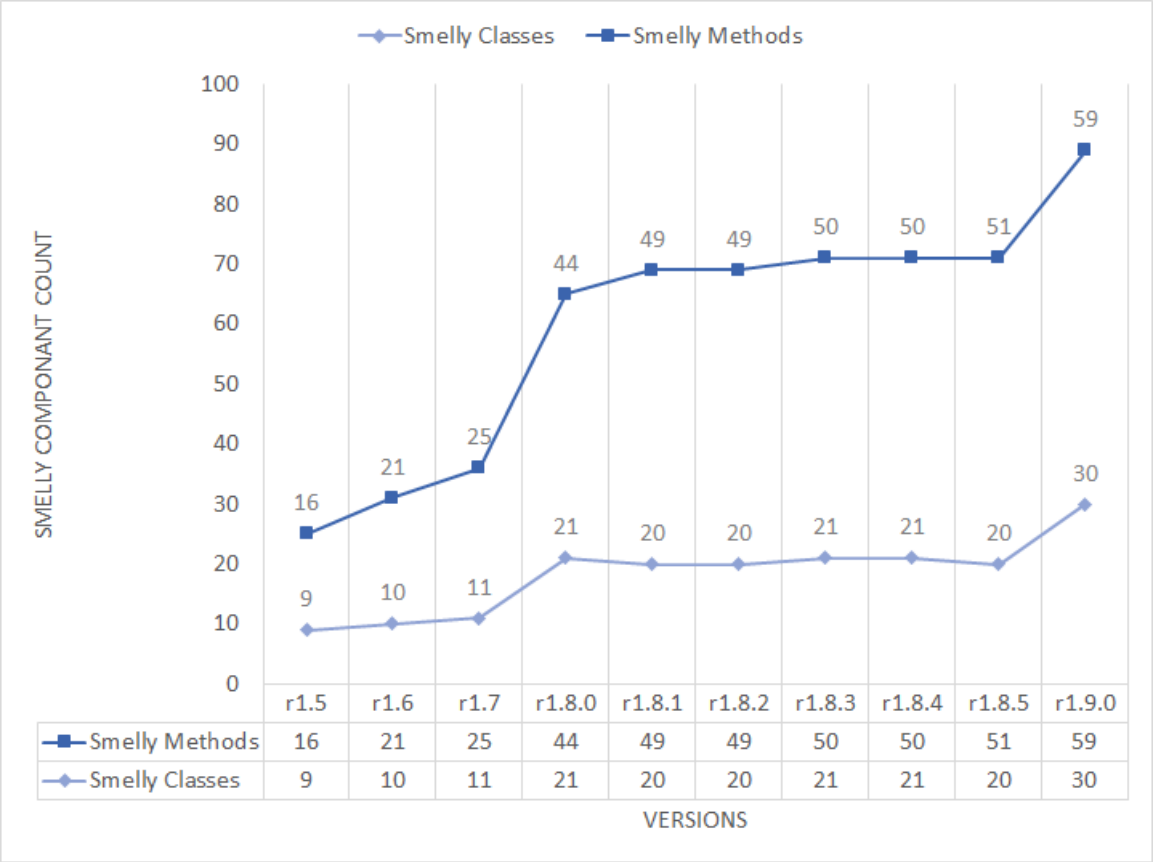


Figure 4.5: Smelly Componants in Mockito

4.4.2 Results: Smell Relation Extraction

In the second stage of the case study, the relation between different smelly components are found. This data is very important as it shows how the connectivity between different instances of code smells change overs time. By analyzing this data, a deeper understanding about the relationship of code smells can be found. From the results of the case study, it is observed that, for all three subjects of the study, number of relations between code smells increases as software grows old.

In Figure 4.7, the number of smell relations for different versions of JUnit, Mockito and Commons-lang is presented. This relationship data was extracted from 25 different release versions. From this figure, it can be seen that after ten releases of

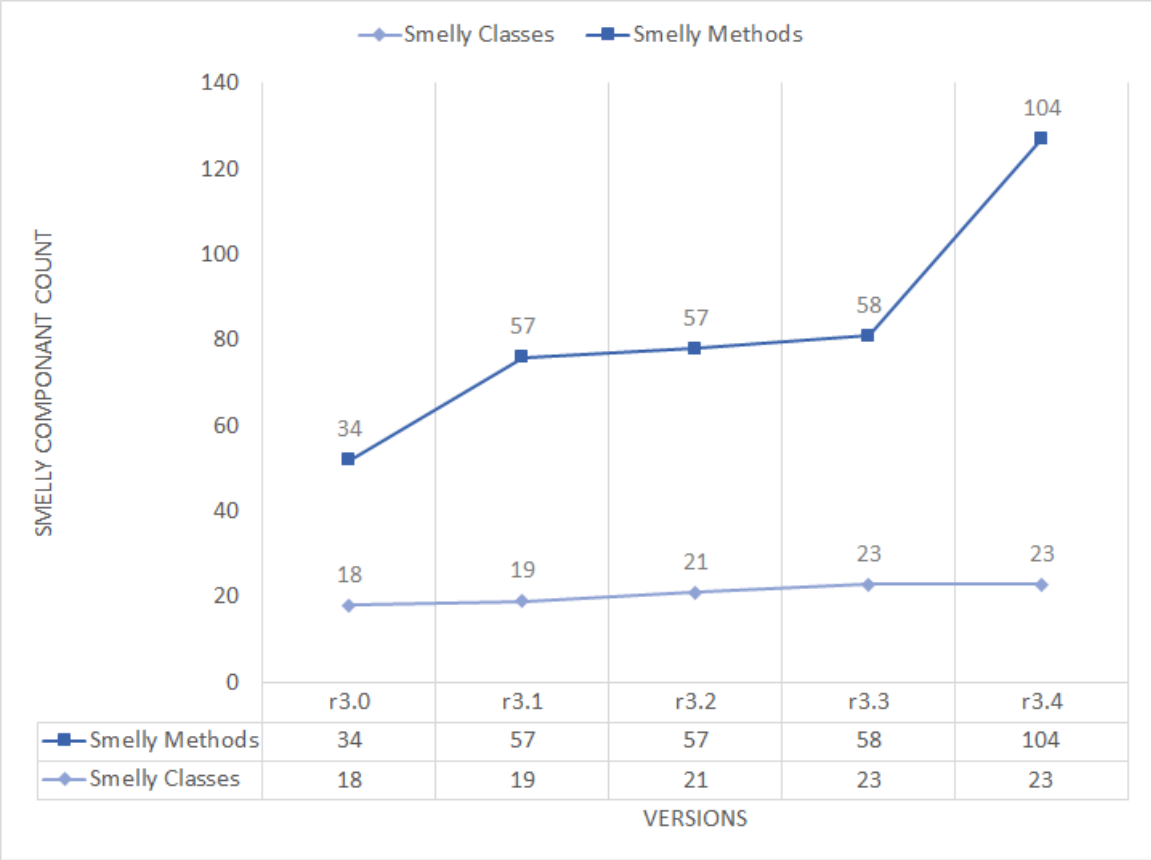


Figure 4.6: Smelly Componants in Commons-lang

JUnit and Mockito, relation count reached 147 from 80, and 81 from 9 respectively. While after five releases of Commons-lang, the relationship count increased from 15 to 90.

4.4.3 Results: Smell Cluster Identification

At this point the clusters of code smells existing in targeted versions of software are found. By comparing clusters from different versions, it can discovered how smell clusters change over time. The found clusters in this step can be divided into two categories (such as - single-node clusters, multiple-node clusters) based on how many nodes those contain. A single-node cluster in this graph, is a smelly component that does not have any architectural connection with any other smelly components. While

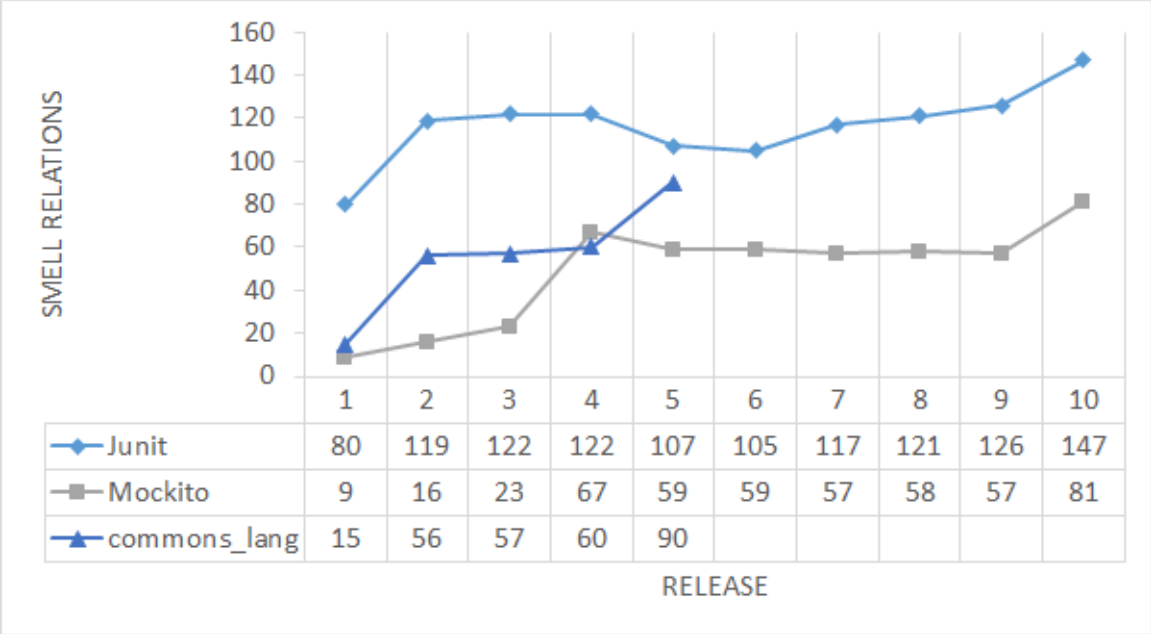


Figure 4.7: Smell Relations for Different Open-Source Software

a multiple-node cluster at least holds two smelly components which are architecturally connected to each other. The found data for all three subjects of this study show that as time passes, number of smell clusters in a software grows steadily.

Graph Properties	r4.5	r4.6	r4.7	r4.8	r4.8.1	r4.8.2	r4.9	r4.10	r4.11	r4.12
Vertex Count	84	95	99	100	101	100	97	102	106	129
Cluster Count	29	21	24	25	33	34	24	25	26	35
Single-Node Count	22	16	20	21	26	28	19	19	19	29
Multiple-Node-Cluster Count	7	5	4	4	7	6	5	6	7	6
Nodes in Multiple-Node-Cluster	62	79	79	79	75	72	78	83	87	100

Figure 4.8: Graph Properties of JUnit

Figure 4.8 shows the detail of graphs generated for each version of JUnit. SCIT found 29 clusters of code smells in JUnit release-4.5. Among these clusters, 22 are single-node and remaining 7 are multiple-node clusters. After ten releases in JUnit-4.12, the cluster count reached 35. Only 6 of the clusters found in this version are actual multiple-node clusters. SCIT also found that, in the final version of JUnit,

100 out of the 129 smelly components were contained inside multiple-node clusters (Figure 4.8).

Graph Properties	r1.5	r1.6	r1.7	r1.8.0	r1.8.1	r1.8.2	r1.8.3	r1.8.4	r1.8.5	r1.9.0
Vertex Count	25	31	36	65	69	69	71	71	71	89
Cluster Count	19	20	19	18	21	21	25	25	26	26
Single-Node Count	17	17	14	12	13	13	18	18	19	18
Multiple-Node-Cluster Count	2	3	5	6	8	8	7	7	7	8
Nodes in Multiple-Node-Cluster	8	14	22	53	56	56	53	53	52	71

Figure 4.9: Graph Properties of Mockito

Figure 4.9 shows the detail of graphs generated for each version of Mockito. SCIT found 19 clusters of code smells in Mockito release-1.5. Among these clusters, 17 are single-node and only 2 are multiple-node clusters. After ten releases in Mockito-1.9.0, the cluster count reached 26. Only 8 of the clusters found in this version are actual multiple-node clusters. SCIT also found that, in the final version of Mockito, 71 out of the 89 smelly components were contained inside multiple-node clusters (Figure 4.9).

Graph Properties	r3.0	r3.1	r3.2	r3.3	r3.4
Vertex Count	52	76	78	81	127
Cluster Count	37	35	36	37	56
Single-Node Count	29	28	28	28	42
Multiple-Node-Cluster Count	8	7	8	9	14
Nodes in Multiple-Node-Cluster	23	48	50	53	85

Figure 4.10: Graph Properties of Commons-lang

Figure 4.10 shows the detail of graphs generated for each version of Commons-lang. SCIT found 37 clusters of code smells in Commons-lang release-3.0. Among these clusters, 29 are single-node and remaining 8 are multiple-node clusters. After five releases in Commons-lang-3.4, the cluster count reached 56. Only 14 of the clusters found in this version are actual multiple-node clusters. SCIT also found

that, in the final version of Commons-lang, 85 out of the 127 smelly components were contained inside multiple-node clusters (Figure 4.10).

4.4.4 Results: Clustering Behavior Inspection

In this final stage of the case study, SKIT provides additional detail about clustering behavior by comparing results of different versions. For example, evolution of cluster size, smell initiation and elimination rate in different cluster, growth rate of different clusters, are provided in the final inspection report. By examining this report, a greater understanding about code smells evolution can be achieved.

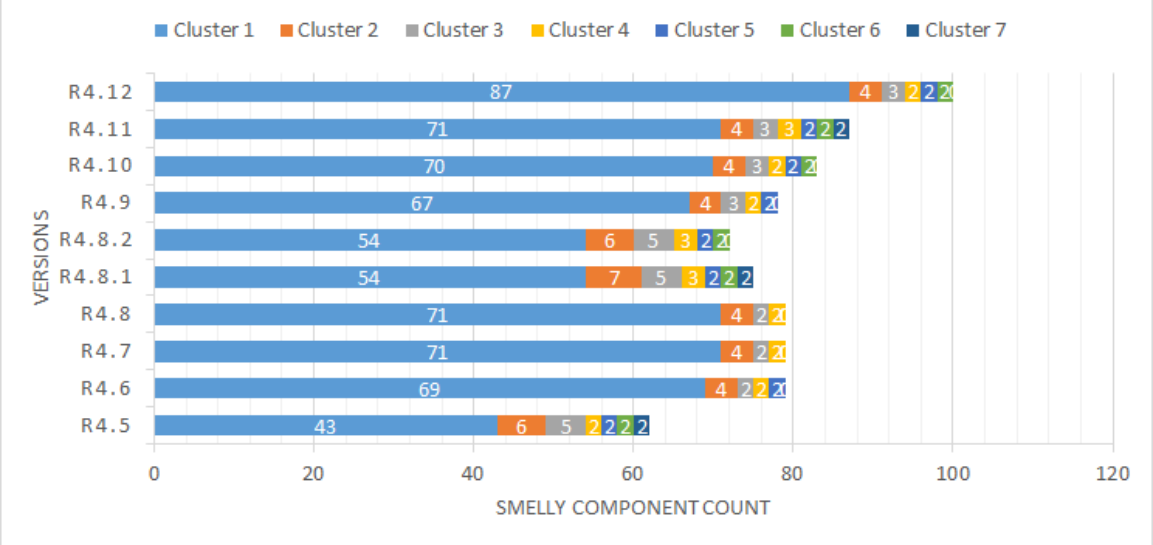


Figure 4.11: Cluster Sizes in JUnit

In Figure 4.11, Evolution of different clusters in JUnit is presented in terms of cluster size. Existence of total 7 smelly clusters are identified in the source code of JUnit. Size of most of these clusters vary from 2 to 7. An extremely large cluster is identified in all ten versions of JUnit. The size of this cluster is between 43 to 87.

In Figure 4.12, Evolution of different clusters in Mockito is presented in terms of cluster size. Existence of total 8 smelly clusters are identified in the source code of

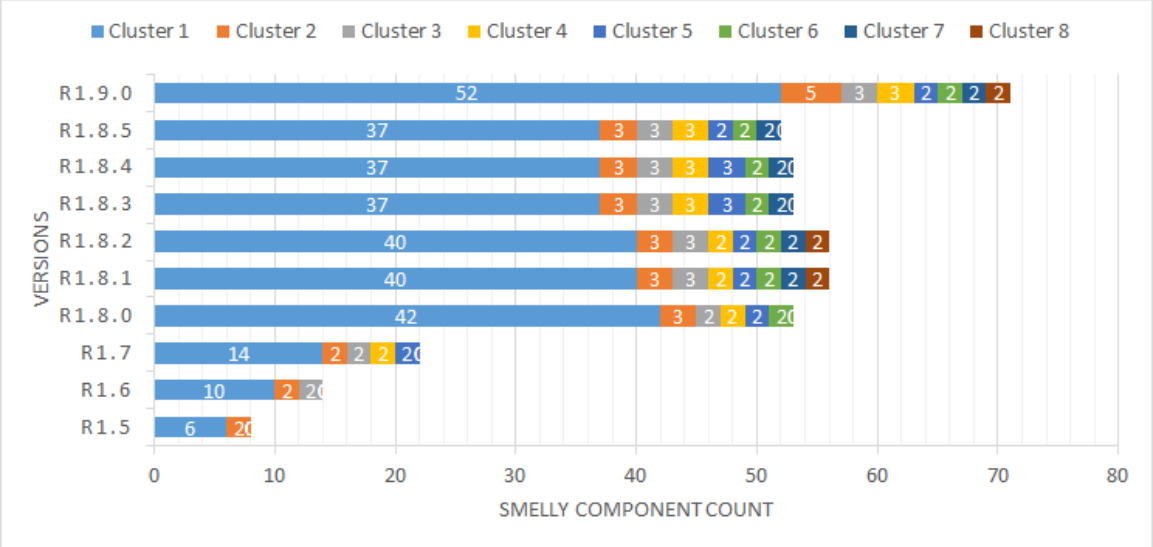


Figure 4.12: Cluster Sizes in Mockito

Mockito. Size of most of these cluster vary from 2 to 5. An extremely large cluster is spotted in all ten versions of Mockito. The size of this cluster is initially 6, however it grew to 52 in the final release.

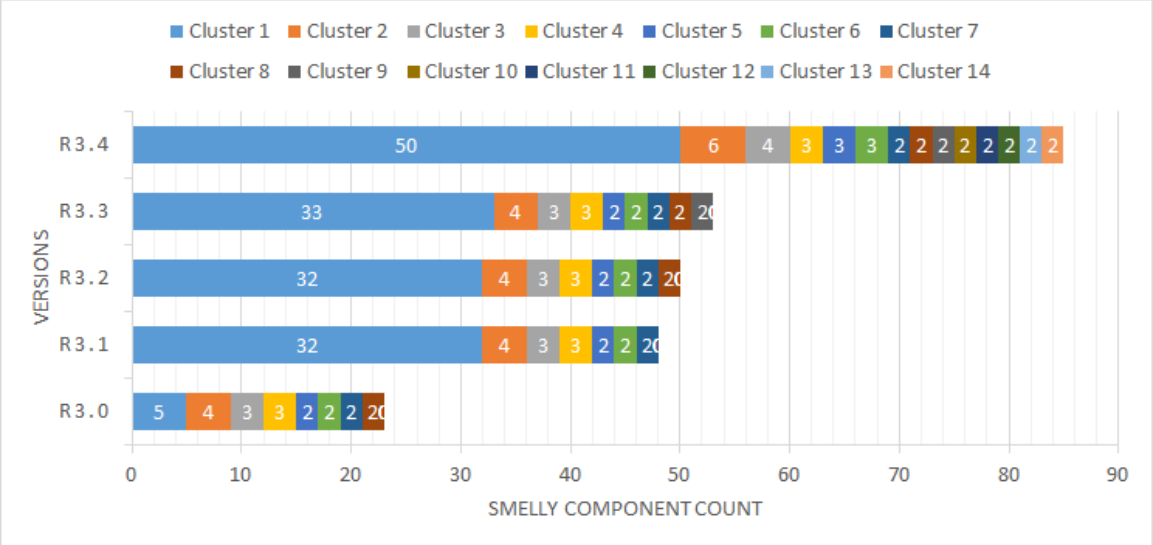


Figure 4.13: Cluster Sizes in Commons-lang

In Figure 4.13, Evolution of different clusters in Commons-lang is presented in terms of cluster size. Existence of total 8 smelly clusters are identified in the source

code of Commons-lang. Size of most of these cluster vary from 2 to 6 nodes. An extremely large cluster is spotted during this study on Commons-lang. The size of this cluster was initially 5, however it eventually grew to 50 in the final release.

4.5 Summery

A tool SCIT was developed that implements the framework FUESC, proposed in the previous chapter, to perform an investigation on evolution of code smell clusters in open-source software. SCIT analyzed 25 release versions of JUnit, Mockito and Apache Commons-lang during this case study. Smell cluster's behavioral and evolutionary properties were extracted from these open source software. It is found that number of smell clusters increases steadily with time. However this increase rate is not equal for all type of clusters. These behaviors will be further investigated in the next chapter to find answers for the research questions of this study.

Chapter 5

Result Analysis

In this chapter, findings of the performed case study are discussed to answer the research questions of this study. From the investigation, empirical evidence has been found that, smelly components get more tightly architecturally connected as time passes. It is also observed that, number of individual smell clusters in software, increases from one version to other. Finally, it is discovered that, code smells tend to create a giant 'Mega Cluster' of smells which is very large in size and difficult to manage.

Earlier in this thesis, three research question are proposed for investigation. Those questions are -

- **RQ1:** How does the architectural connectivity of smelly components evolve with time?
- **RQ2:** How does the existence of code smell clusters change over time?
- **RQ3:** Do all the smell clusters show similar pattern of evolution?

A framework named FUESC is proposed in Chapter 3 to analyze evolution patterns of code smells. Based on this framework, a smell cluster inspection tool SCIT

was developed to study clustering behavior of real-life software. SKCT was used to perform a case study on 25 versions of three popular java libraries (which are JUnit, Mockito and Commons-lang). The results of the above mentioned case study is inspected in this chapter to answer the research questions. The following three subsections will focus on answering these research questions one by one.

5.1 RQ1: Total number of architectural connections between smells increases steadily with time

How does the architectural connectivity of smelly components evolve with time?

The first goal of this investigation is to understand how architectural connectivity of code smells evolve in a software. After analyzing the results of the case study, it is discovered that relationship between code smells increases as a software grows old. The relationship count raises in a steady pace from one version to the next. All three subjects of this case study (which are JUnit, Mockito and Commons-lang) exhibited this behavior.

For JUnit, the smell relation count in release r4.5 was 80 which grew to 147 in the final release r4.12. An average increase of 7.44 (Figure 5.1) relations per release was discovered. In each new release the relation count is increased by 7% (Figure 5.2) on an average. Mockito and Commons-lang also showed similar patterns of change in relation count. This count only grew with time in both of this projects. Average increase for Mockito is 8 relations per release, where it is 18.75 for Commons-lang as shown in Figure 5.1. The growth rate is calculated at 28% and 57% respectively for

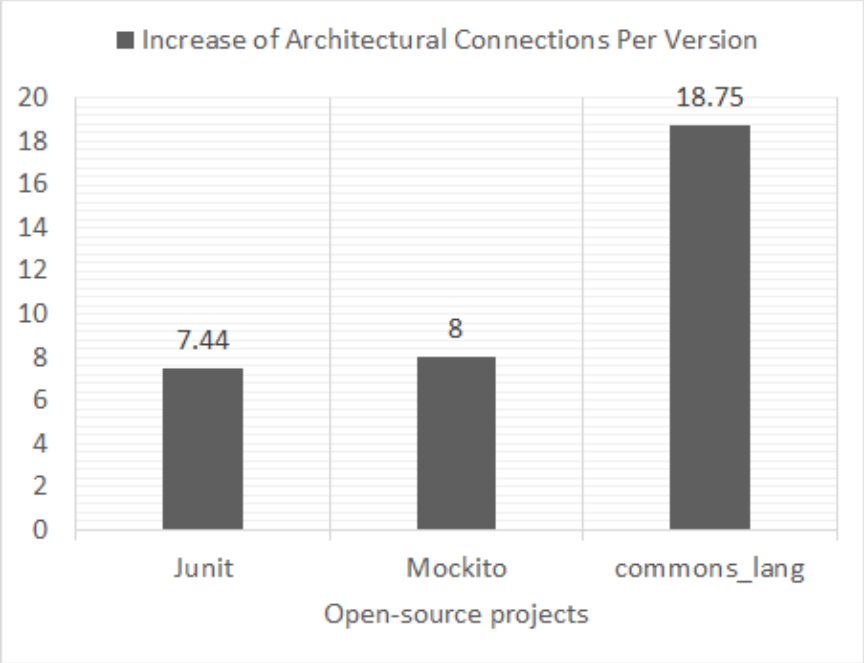


Figure 5.1: Average Increase

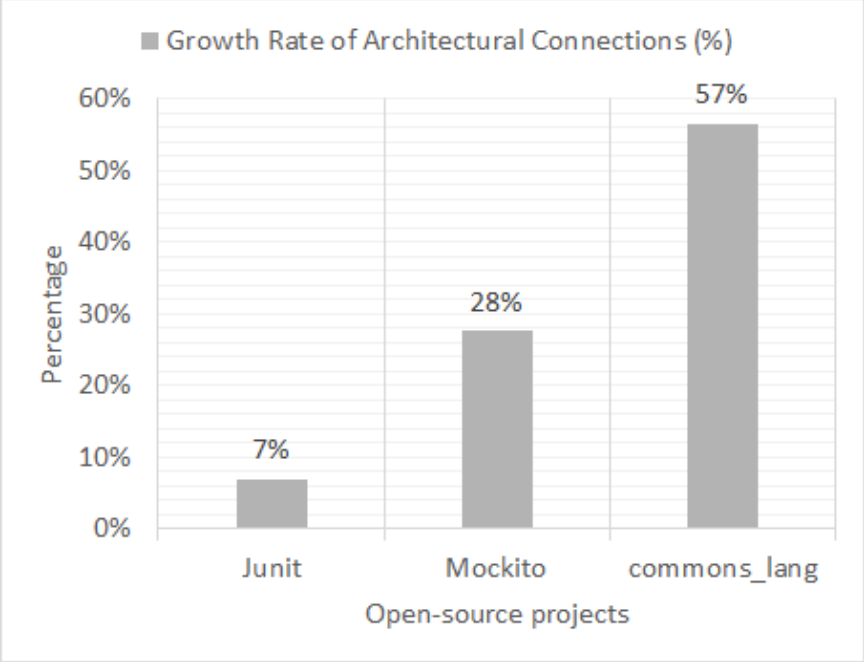


Figure 5.2: Growth Rate

Figure 5.3: Increase of architectural relations in different software

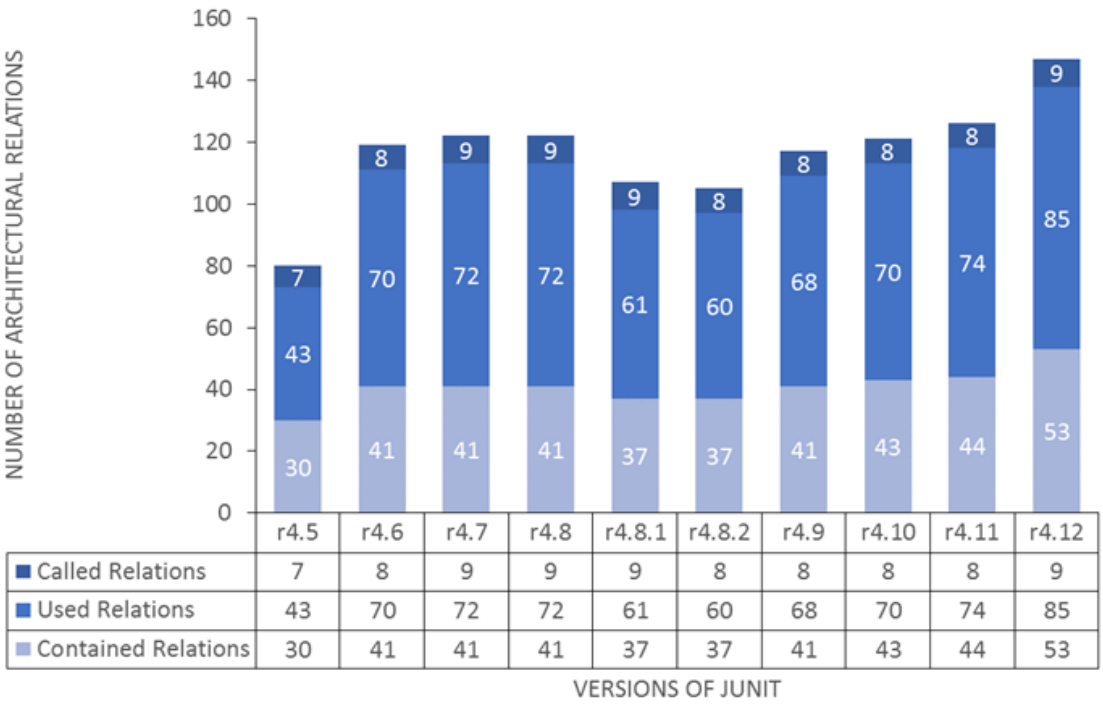


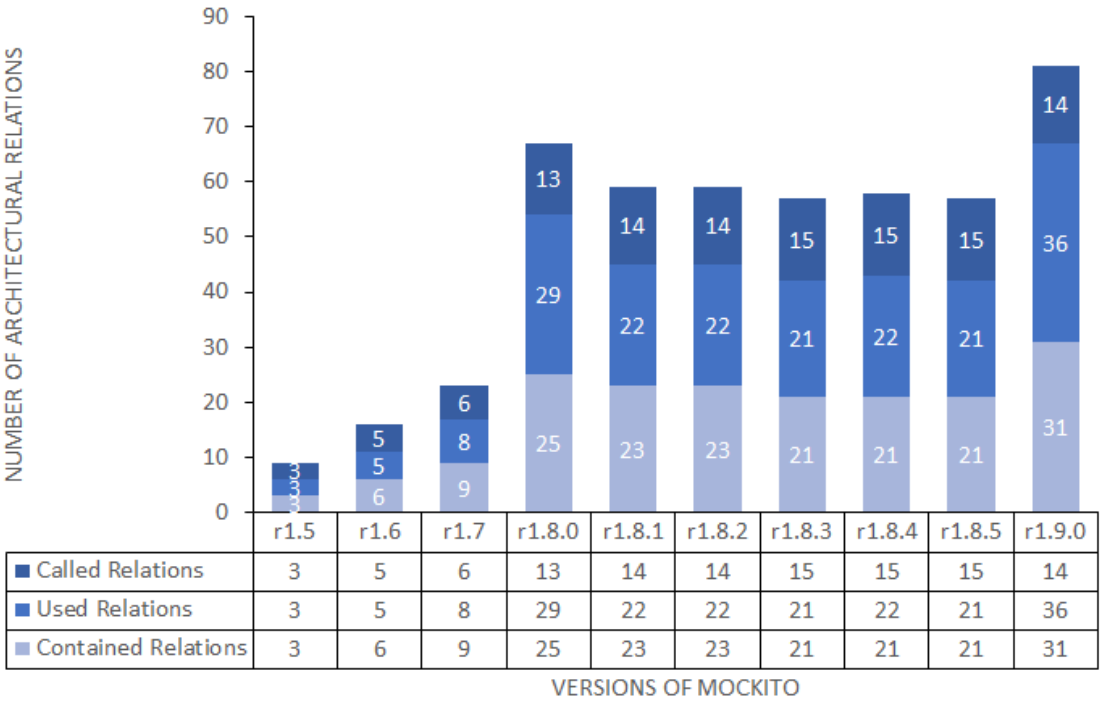
Figure 5.4: Number of architectural relations in different versions of JUnit

Mockito and Commons-lang (Figure 5.2).

As discussed in chapter 2, there are three types of code smell relations such as Contained Relation, Used Relation and Called Relation. So far it has been shown how the total number of relationship changed with time. Next it will be shown how different type of relationship count evolves as time passes.

As shown in Figure 5.4, Called Relation count of JUnit4.5 is 7 which grew to 9 in JUnit4.12 after 10 releases. Number of Used Relation raised to 85 from 43 in this period, where Contained Relationship count reached 53 from 30. All three kinds of relationship increased during these releases as the total relation count increased to 147 from 80.

As shown in Figure 5.5, Called Relation count of Mockito1.5 is 3 which grew to 14 in Mockito1.9.0 after 10 releases. Number of Used Relation raised to 36 from 3 in this period, where Contained Relationship count reached 31 from 3. All three kinds



	r1.5	r1.6	r1.7	r1.8.0	r1.8.1	r1.8.2	r1.8.3	r1.8.4	r1.8.5	r1.9.0
Called Relations	3	5	6	13	14	14	15	15	15	14
Used Relations	3	5	8	29	22	22	21	22	21	36
Contained Relations	3	6	9	25	23	23	21	21	21	31

Figure 5.5: Number of architectural relations in different versions of Mockito

of relationship increased during these releases as the total relation count reached 81 from 9.

As shown in Figure 5.6, Called Relation count of Commons-lang3.0 is 5 which grew to 25 in Commons-lang3.4 after 5 releases. Number of Used Relation raised to 7 from 19 in this period, where Contained Relationship count reached 46 from 3. All three kinds of relationship increased during these releases as the total relation count raised to 90 from 15.

Now, this behavior of software source code indicates that code smells residing in a software tends to be inter-connected. As time passes interaction between smelly classes and smelly methods increase in a steady manner. It means, if a code segment infected by code smells is not refactored, there is a good chance that it will connect to other smelly components in future releases of the software and eventually make it even more unmanageable.

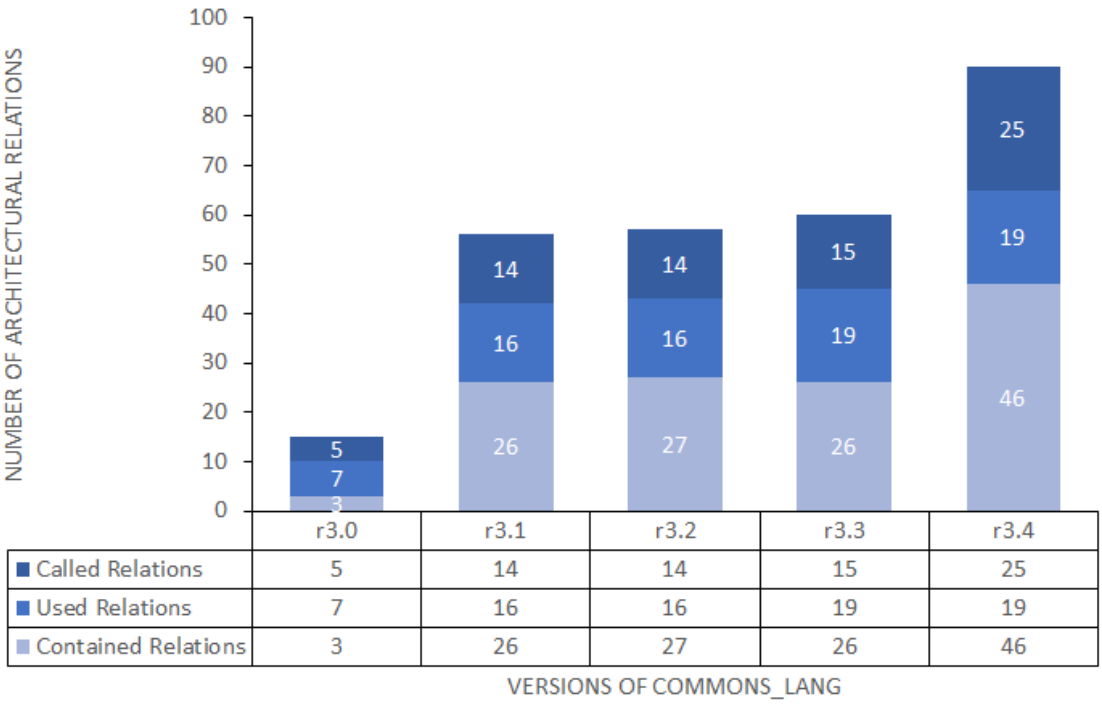


Figure 5.6: Number of architectural relations in different versions of Commons-lang

5.1.1 'Contained Relationship' count increases steadily with time

From Figure 5.4, 5.5 and 5.6, it can be observed that the total number of Contained Relationship between smells raises with time. To understand this behavior, average increase and growth rate were calculated for JUnit, Mockito and Commons-lang. It is found that Contained Relation count for JUnit increase at a steady pace of 2.56 relations per release. The increment for Mockito and connons-lang is calculated at 3.11 and 10.75 respectively (Figure 5.7). Commons-lang exhibits the highest growth rate of 98% per release as shown in Figure 5.8.

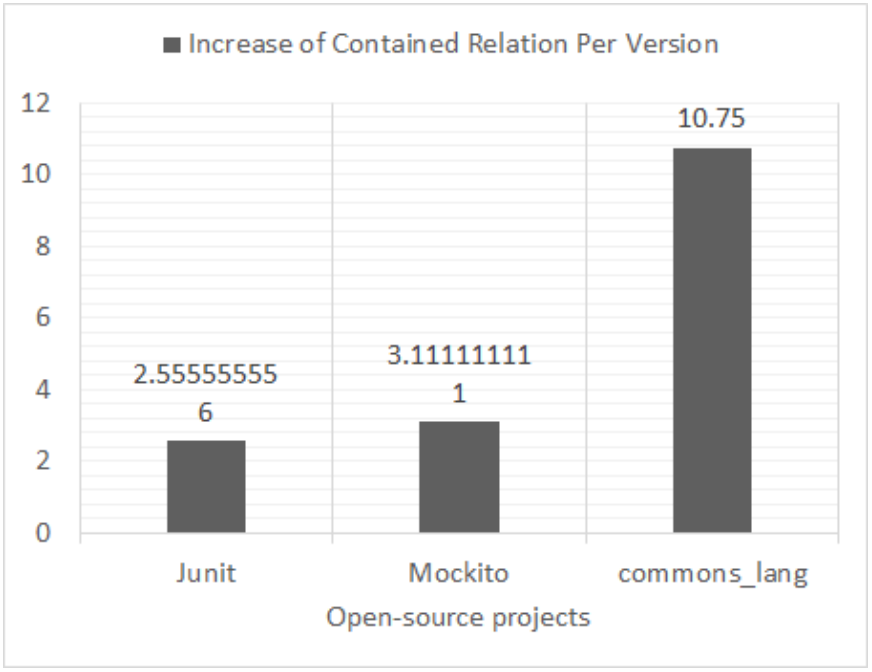


Figure 5.7: Average increase in Contained Relation count

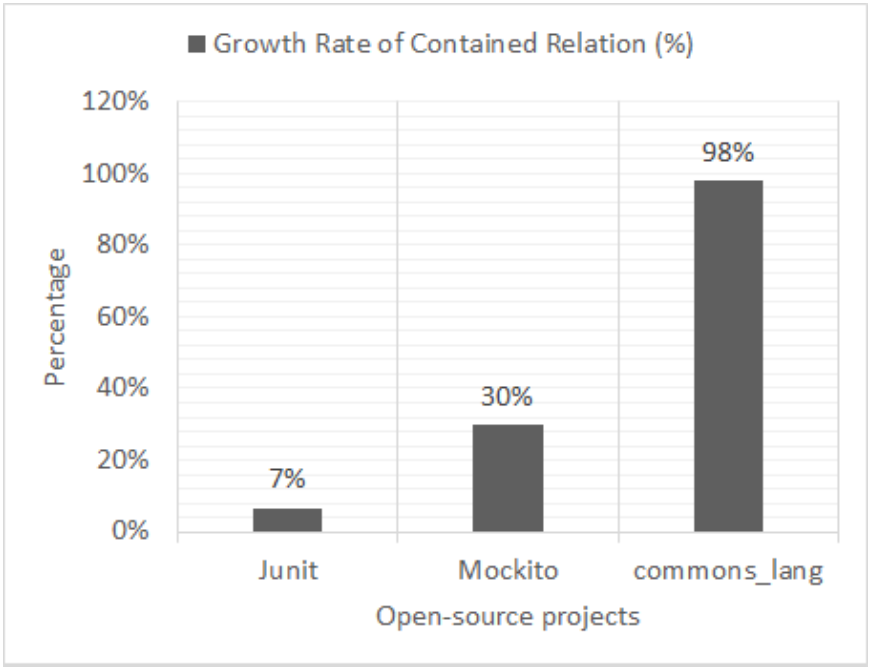


Figure 5.8: Growth rate of Contained Relationship

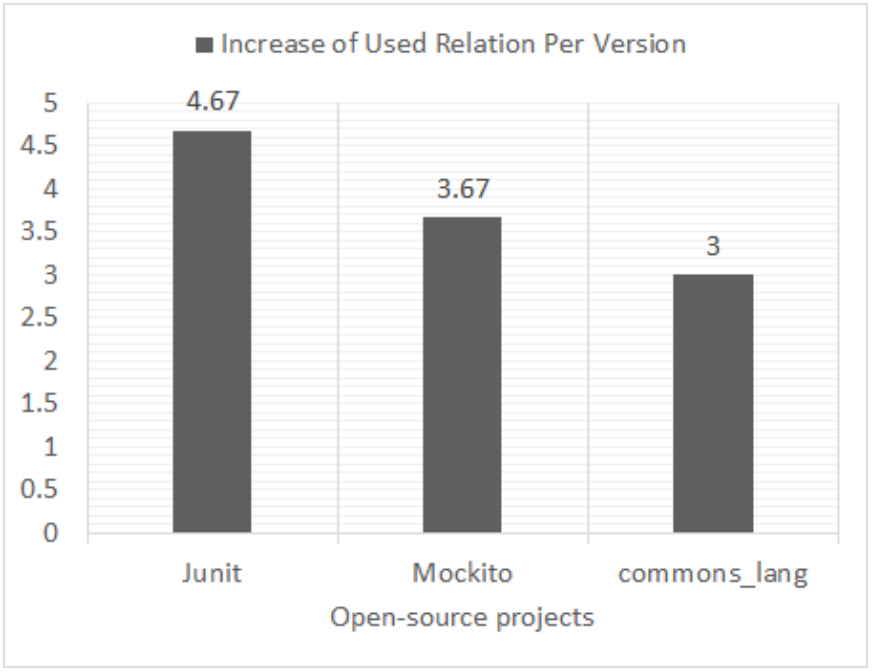


Figure 5.9: Average increase in Used Relation count

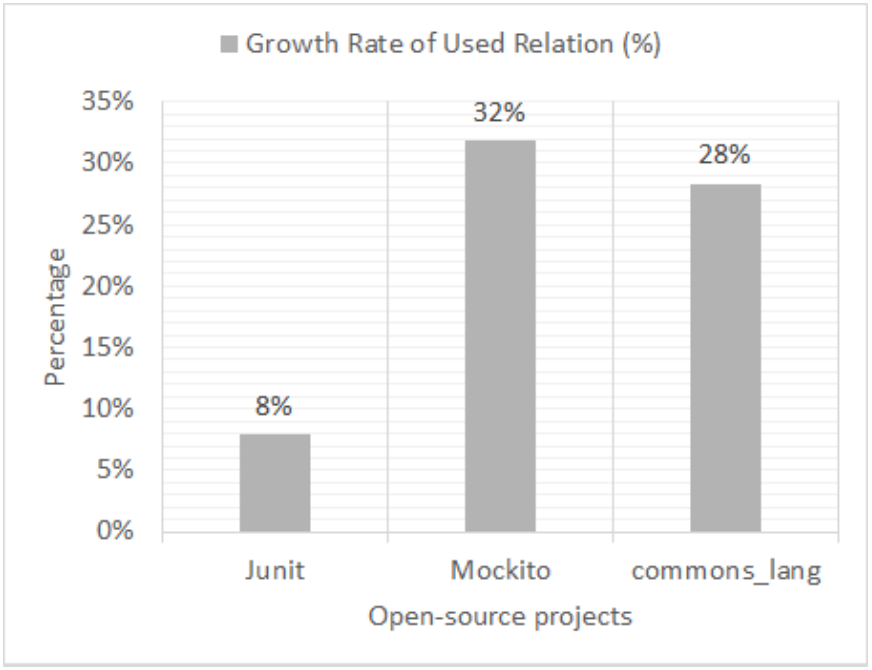


Figure 5.10: Growth rate of Used Relationship

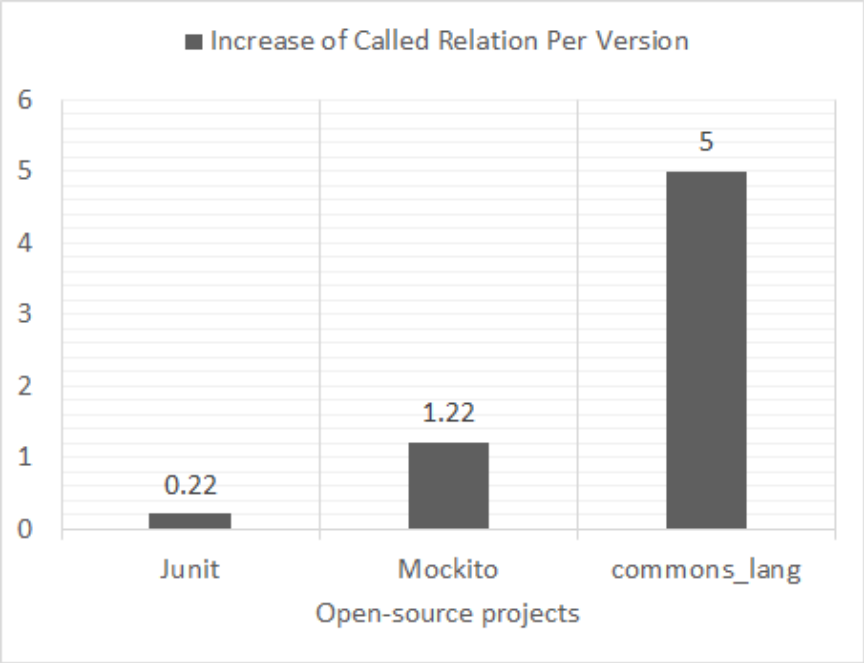


Figure 5.11: Average increase in Called Relation count

5.1.2 'Used Relationship' count increases steadily with time

From Figure 5.4, 5.5 and 5.6, it can be observed that the total number of Used Relationship between smells raises with time. To understand this behavior, average increase and growth rate were calculated for JUnit, Mockito and Commons-lang. It is found that Used Relation count for JUnit increase at a steady pace of 4.67 relations per release. The increment for Mockito and connons-lang is calculated at 3.67 and 3 respectively (Figure 5.9). Mockito exhibits the highest growth rate of 32% per release as shown in Figure 5.10.

5.1.3 'Called Relationship' count increases steadily with time

From Figure 5.4, 5.5 and 5.6, it can be observed that the total number of Called Relationship between smells raises with time. To understand this behavior, average increase and growth rate were calculated for JUnit, Mockito and Commons-lang. It

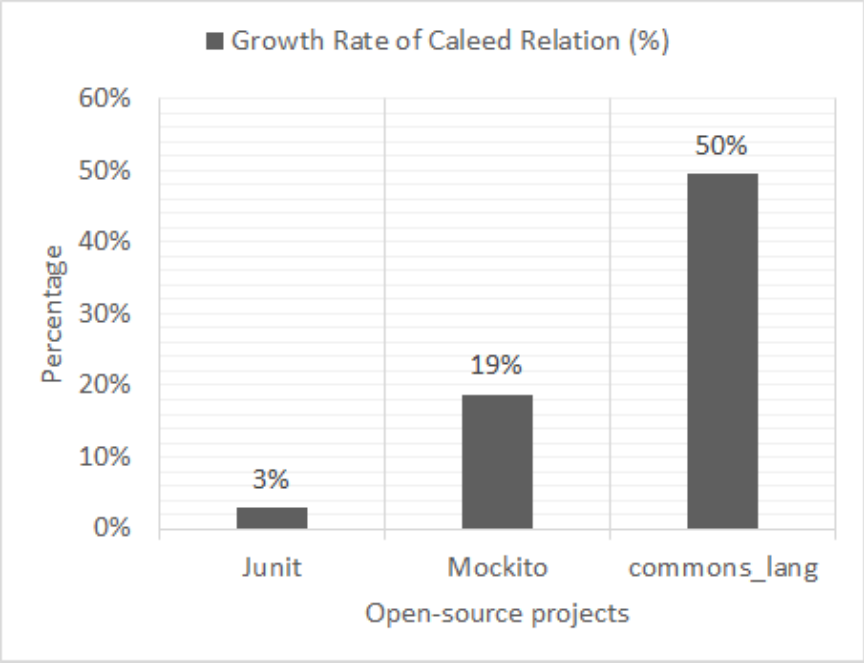


Figure 5.12: Growth rate of Called Relationship

is found that Used Relation count for JUnit increase at a steady pace of 0.22 relations per release. The increment for Mockito and Commons-lang is calculated at 1.22 and 5 respectively (Figure 5.11). Commons-lang exhibits the highest growth rate of 50% per release as shown in Figure 5.12.

5.2 RQ2: Total number of code smell cluster in a software increases steadily with time

How does the existence of code smell clusters change over time?

The second research goal of this investigation is to understand how the existence of different clusters change with time. After analyzing the results of the case study, it is discovered that the number of code smell clusters existing in the source code increases as the software grows older. It is also found that the cluster count increases from one version to the next in a steady pace. All three subjects of this case study

PROJECTS	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
Junit	29	21	24	25	33	34	24	25	26	35
Mockito	19	20	19	18	21	21	25	25	26	26
Commons-lang	37	35	36	37	56					

Figure 5.13: Cluster count for different releases of open-source software

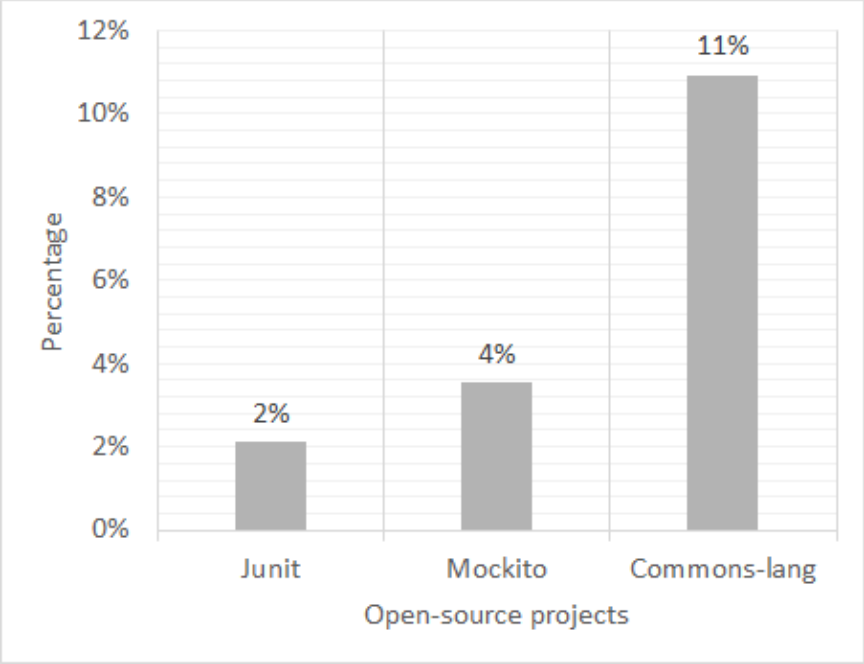


Figure 5.14: Cluster Count Growth Rate

(which are JUnit, Mockito and Commons-lang) exhibites this behavior.

For JUnit, the cluster count in release r4.5 was 29 which grew to 35 in the final release r4.12 as shown in Table 5.13. An average increase of 0.67 clusters per release was discovered. In each new release the cluster count is increased by 2% on an average. Mockito and Commons-lang also showed similar patterns of change in cluster count. This count only grew with time in both of this projects. Average increment for Mockito is 0.78 clusters per release, where it is 4.75 for Commons-lang. The growth rate is calculated at 4% and 11% respectively for Mockito and Commons-lang (Figure 5.14).

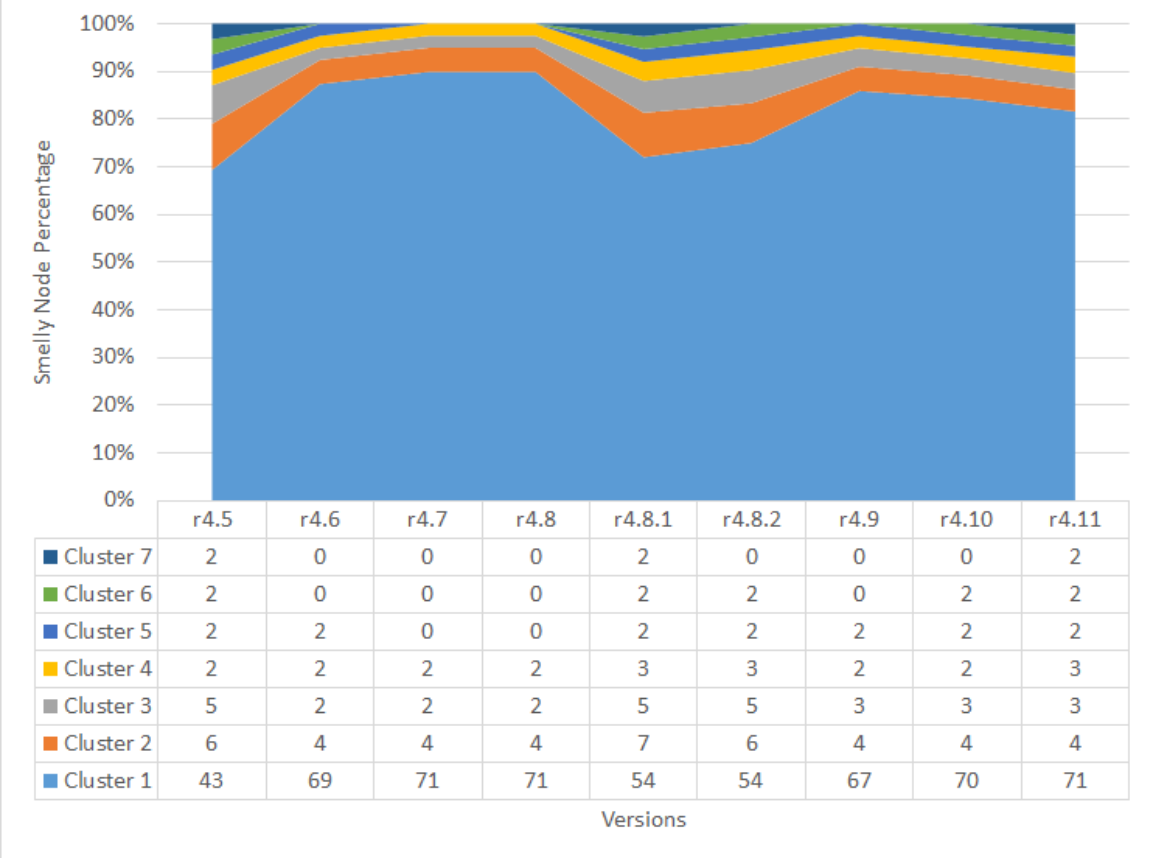
Now, this behavior of software source code indicates that code smells residing in a software tends to create clusters rather than staying as isolated instances. As time passes interaction between smelly classes and smelly methods increase in a steady manner and increase the number of smell cluster. Which means, if a code segment infected by code smells is not refactored, there is a good chance that it will connect to other smelly components in future releases of the software and eventually larger clusters of bad smells.

5.3 RQ3: Smelly Components Tend to Create a 'Mega Cluster' of Bad Smells

All the smelly clusters in source code do not exhibit the same patterns of evolution. Every single cluster's size in JUnit is presented in Figure 5.15. Here, it can be seen that a large cluster contains the lion's share of all smelly components. The percentage of the smells it holds is very consistent and always over 70%. Its size grew from 43 (release-4.5) to 71 (release-4.12) in ten releases. While other clusters are significantly smaller in size with the average size of 2.35 nodes.

Similar clustering behavior is observed in Mockito as well. Every single cluster's size in Mockito is presented in Figure 5.16. Here, it can be seen that a large cluster contains maximum number of smelly components. The percentage of the smells it holds is very consistent and always over 60%. Its size grew from just 6 (release-1.5) to 52 (release-1.9.0) in ten releases. While other clusters are significantly smaller in size with the average size of 1.89 nodes.

Similar clustering behavior is seen in Commons-lang as well. Every single cluster's size in Commons-lang is presented in Figure 5.17. Here, it can be seen that a large cluster contains maximum number of smelly components. The percentage of the



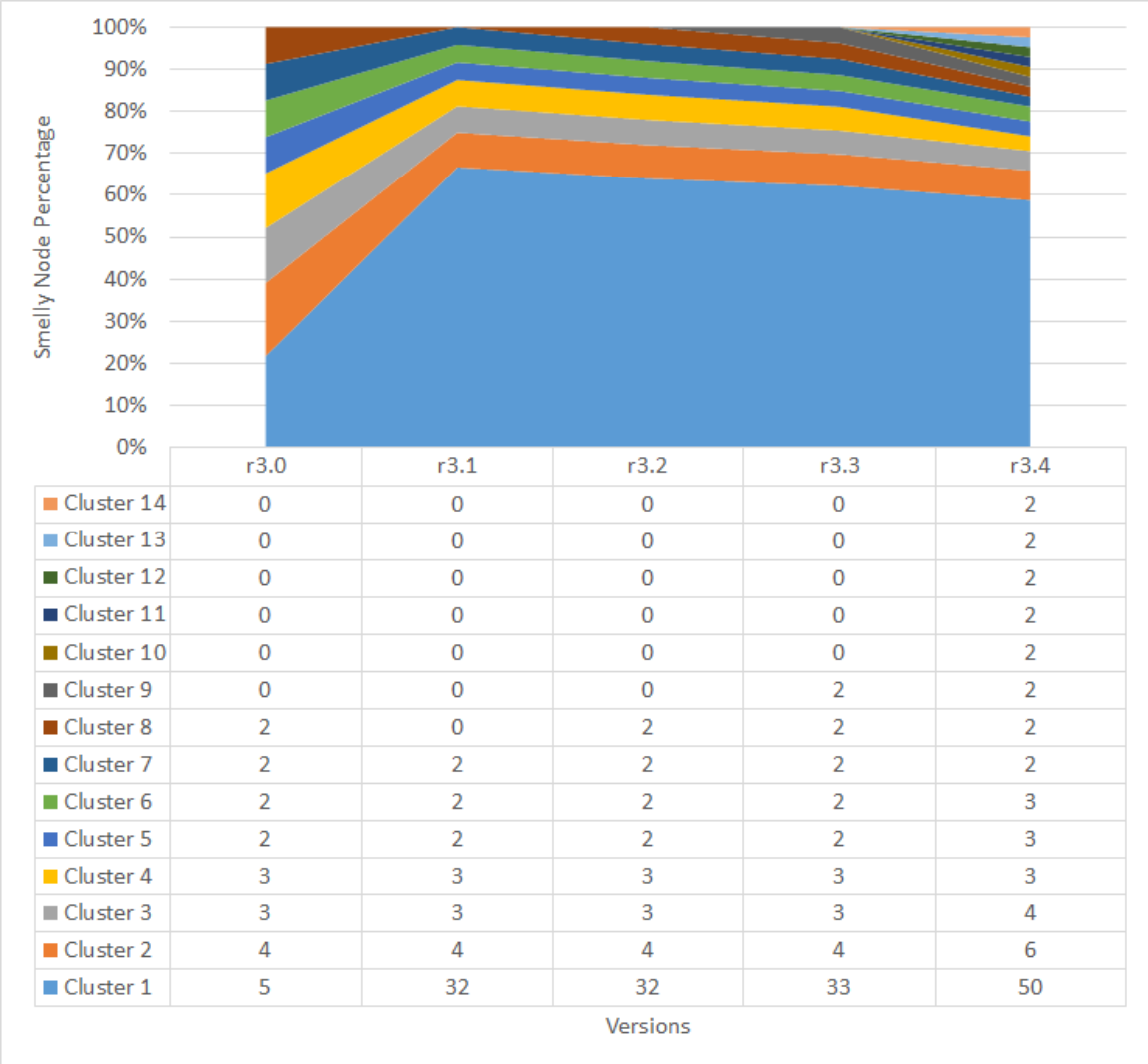
	r4.5	r4.6	r4.7	r4.8	r4.8.1	r4.8.2	r4.9	r4.10	r4.11
Cluster 7	2	0	0	0	2	0	0	0	2
Cluster 6	2	0	0	0	2	2	0	2	2
Cluster 5	2	2	0	0	2	2	2	2	2
Cluster 4	2	2	2	2	3	3	2	2	3
Cluster 3	5	2	2	2	5	5	3	3	3
Cluster 2	6	4	4	4	7	6	4	4	4
Cluster 1	43	69	71	71	54	54	67	70	71

Figure 5.15: Evolution of smell clusters in JUnit

	r1.5	r1.6	r1.7	r1.8.0	r1.8.1	r1.8.2	r1.8.3	r1.8.4	r1.8.5	r1.9.0
Cluster 8	0	0	0	0	2	2	0	0	0	2
Cluster 7	0	0	0	0	2	2	2	2	2	2
Cluster 6	0	0	0	2	2	2	2	2	2	2
Cluster 5	0	0	2	2	2	2	3	3	2	2
Cluster 4	0	0	2	2	2	2	3	3	3	3
Cluster 3	0	2	2	2	3	3	3	3	3	3
Cluster 2	2	2	2	3	3	3	3	3	3	5
Cluster 1	6	10	14	42	40	40	37	37	37	52

Versions

Figure 5.16: Evolution of smell clusters in Mockito



	r3.0	r3.1	r3.2	r3.3	r3.4
Cluster 14	0	0	0	0	2
Cluster 13	0	0	0	0	2
Cluster 12	0	0	0	0	2
Cluster 11	0	0	0	0	2
Cluster 10	0	0	0	0	2
Cluster 9	0	0	0	2	2
Cluster 8	2	0	2	2	2
Cluster 7	2	2	2	2	2
Cluster 6	2	2	2	2	3
Cluster 5	2	2	2	2	3
Cluster 4	3	3	3	3	3
Cluster 3	3	3	3	3	4
Cluster 2	4	4	4	4	6
Cluster 1	5	32	32	33	50

Versions

Figure 5.17: Evolution of smell clusters in Commons-lang

smells it holds is grew steadily to over 60%. Its size grew from just 6 (release-3.0) to 50 (release-3.4) in five releases. While other clusters are significantly smaller in size with the average size of 2.30 nodes.

5.3.1 Smells tend to create a 'Mega Cluster' over time

Existence of a giant cluster of code smells is found in this study. This cluster is identified as 'Mega Cluster'. The 'Mega Cluster' has few unique behaviors in terms

PROJECTS	r4.5	r4.6	r4.7	r4.8	r4.8.1	r4.8.2	r4.9	r4.10	r4.11	r4.12
In Single-Nodes	0	2	3	1	1	2	0	0	5	14
In Small Clusters	0	1	0	0	3	0	0	2	2	3
In the Largest Cluster	0	10	2	0	0	0	5	3	2	16

Figure 5.18: Initiation of code smells in different clusters of JUnit

of size and growth which separate it from other clusters. The mega cluster found in JUnit has a growth rate of 8% while no other cluster in this project have a positive growth rate. In mockito, the growth rate is calculated at 27%. the only other growth rate is 11%, which is of a cluster that grew from 2 to 5 nodes in ten releases. In Commons-lang, the growth rate of the mega cluster is staggering 78%. A few other clusters have 11% growth rate, all of those having less than 7 nodes. The average percentage of nodes contained by 'Mega Clusters' is also very distinct. For JUnit this rate is 82.23%. While it is 71.62% and 54.70% for Mockito and Commons-lang respectively. This figures confirm that, the 'Mega Cluster' exhibits a different pattern of evolution from all other clusters.

5.3.2 Code smell initiation rate is relatively high in 'Mega Clusters'

To further examine the behavior of 'Mega Clusters' the code smell initiation rate in different clusters is analyzed. It is found in all three systems that, the initiation rate of code smells is higher in 'Mega Clusters'.

In Table 5.18, all the instances of code smell initiation in JUnit is presented. These instances are divided in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is highest with 38 instances. Initiation of 10 new instances at a time was registered in release-4.6.

In Table 5.20, all the instances of code smell initiation in Commons-lang is pre-

PROJECTS	r1.5	r1.6	r1.7	r1.8.0	r1.8.1	r1.8.2	r1.8.3	r1.8.4	r1.8.5	r1.9.0
In Single-Nodes	0	7	0	9	3	0	5	0	1	4
In Small Clusters	0	2	3	9	3	0	4	0	0	7
In the Largest Cluster	0	3	4	21	1	0	2	0	0	11

Figure 5.19: Initiation of code smells in different clusters of Mockito

PROJECTS	r3.0	r3.1	r3.2	r3.3	r3.4
In Single-Nodes	0	4	3	0	15
In Small Clusters	0	3	2	2	14
In the Largest Cluster	0	17	0	2	17

Figure 5.20: Initiation of code smells in different clusters of Commons-lang

sented in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is highest with 42 instances. Initiation of 21 new instances at a time was registered in release-1.8.0.

In Table 5.19, all the instances of code smell initiation in Mockito is presented in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is highest with 36 instances. Initiation of 17 new instances at a time was registered in release-3.1 and release-3.4.

Similar behavior is observed in each of the software in terms of smell initiation. A comparison of smell initiation between JUnit, Mockito and Commons-lang is shown in Table 5.21. In all three of these software, the highest number of smells are initiated in the Mega Cluster with percentage varying between 42-49%.

PROJECTS	In Single Nodes	In Small Clusters	In the Largest Cluster	In the Largest Cluster(%)
Junit	28	11	38	49%
Mockito	29	28	42	42%
Commons-lang	22	21	36	46%

Figure 5.21: Initiation of code smells in different software

PROJECTS	r4.5	r4.6	r4.7	r4.8	r4.8.1	r4.8.2	r4.9	r4.10	r4.11	r4.12
In Single-Nodes	0	1	0	0	0	1	6	0	4	3
In Small Clusters	0	1	1	0	0	2	1	0	0	5
In the Largest Cluster	0	0	0	0	3	0	1	0	1	2

Figure 5.22: Elimination of code smells in different clusters of JUnit

PROJECTS	r1.5	r1.6	r1.7	r1.8.0	r1.8.1	r1.8.2	r1.8.3	r1.8.4	r1.8.5	r1.9.0
In Single-Nodes	0	6	2	6	0	0	1	0	0	1
In Small Clusters	0	0	0	3	0	0	4	0	1	3
In the Largest Cluster	0	0	0	1	3	0	4	0	0	0

Figure 5.23: Elimination of code smells in different clusters of Mockito

5.3.3 Code smell elimination rate is relatively low in 'Mega Clusters'

In Table 5.22, all the instances of code smell elimination in JUnit is presented. These instances are divided in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is lowest with 7 instances. Elimination of 3 smells at a time was registered in release-4.8.1.

In Table 5.24, all the instances of code smell elimination in Commons-lang is presented in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is lowest with just 8 instances. Elimination of 4 smells at a time was registered in release-1.8.3.

In Table 5.23, all the instances of code smell elimination in Mockito is presented in three categories, instances in single-node cluster, instances in the Mega Cluster and instances in all other small clusters. It can be seen that, the number of instances in the largest cluster is lowest with just 1 instance. This instance was registered in release-3.3.

Similar behavior is observed in each of the software in terms of smell elimination.

PROJECTS	r3.0	r3.1	r3.2	r3.3	r3.4
In Single-Nodes	0	0	3	0	0
In Small Clusters	0	0	0	0	0
In the Largest Cluster	0	0	0	1	0

Figure 5.24: Elimination of code smells in different clusters of Commons-lang

PROJECTS	In Single Nodes	In Small Clusters	In the Largest Cluster	In the Largest Cluster(%)
Junit	15	10	7	22%
Mockito	16	11	8	23%
Commons-lang	3	0	1	25%

Figure 5.25: Elimination of code smells in different software

A comparison of smell elimination between JUnit, Mockito and Commons-lang is shown in Table 5.25. In all three of these software, the lowest number of smells are eliminated in the Mega Cluster with percentage varying between 22-25% which is almost half the rate of smell initiation (which is 42-49%).

5.4 Summery

After analyzing the results, decisive answers can be found for the research questions that are proposed earlier in this study. It is found that the number of relations between code smells in JUnit, Mockito and Commons-lang increased steadily with time with the growth rate of 7%, 28% and 57% respectively. It is also seen that the cluster count for these software also increased at a steady pace which is 2% for JUnit, 4% for Mockito and 11% for Commons-lang. Finally, the existence of a giant cluster is found in all three of these software. These giant cluster hold more than 50% of all the smelly components in these software and have a relatively higher growth rate than other clusters. These clusters are named define as 'Mega Clusters'. The percentage of new smell initiation in these Mega Clusters vary from 42-49%, while the percentage of smell elimination vary between 22-25%. So the overall result indicated that, if

code smells are not refactored in code, these will architecturally connect with other smells in the system, create clusters of code smells, and eventually form a giant 'Mega Cluster' in which, smell initiation is more likely to happen than smell elimination.

Chapter 6

Conclusion and Future Direction

A framework named FUESC is proposed in this research to analyze evolution patterns of code smell clusters. Based on this framework, a smell cluster inspection tool SCIT is developed to study clustering behavior in real-life systems. SCIT is used to perform a case study on 25 versions of three popular java libraries (which are JUnit, Mockito and commons-lang). After analyzing the results, decisive answers can be found for the research question that have been proposed earlier in this study. It is found that, for JUnit, Mockito and Commons-lang, the number of architectural relations between code smells increases steadily with time, with the growth rate of 7&, 28% and 57% respectively. It is also seen that the cluster count for these software also increases at a steady pace which is 2% for JUnit, 4% for Mockito and 11% for Commons-lang. Finally, the existence of a giant cluster is found in all three of these software. These giant cluster holds more that 50% of all the smelly components in these software, and have a relatively higher growth rate than other clusters. The percentage of new smell initiation in these clusters vary from 42-49%, while the percentage of smell elimination vary between 22-25%. So the overall result indicated that, if code smells are not refactored in code, these will architecturally connect with other smells in the

system, create clusters of code smells, and eventually form a giant 'Mega Cluster' in which, smell initiation is more likely to happen than smell elimination.

6.1 Threads to Validity

The internal and external validity of this study is discussed in this section.

6.1.1 External Validity

In this study is restricted to open-source libraries which are hosted in Github. Therefore, it cannot be claimed that the findings of this study will apply to commercial projects. As all the subject of this study are written in java, it cannot be guaranteed that projects written in different languages exhibits similar behavior. However, there are not much differences in coding practice between java and other object oriented languages. In addition, the projects used in this study (JUnit, Mockito and Common-lang) are among the most popular libraries in Java.

In a software project, coding practice, change behavior and project structure highly depend on the type of project. These factors are not considered in this study to get more generalized findings. Developers level of expertise, is not taken into account to limit the scope of this research.

6.1.2 Internal Validity

It is a common believe that, human intuition is the best way to detect code smells. That is why, there is still no accepted standard for detecting smells. However, manually smell detection is time consuming and has inconsistent accuracy. That is why this study relies on JUnit for detecting smells. Therefore, the outcome of this study may vary depending on precision and recall of JUnit.

This investigation is restricted to four code smells. But there exist many others. A future study targeting all the well known smells will help to gather a more generalized finding.

In real-life software development sometimes the developers restructure the projects for better understanding. The proposed framework does not automatically identify project restructuring as that work is in a completely different scope. To minimize the impact of project restructuring, a manual inspection is performed which identified the versions with most similar project structure.

Sometimes certain methods or classes in version n are renamed in version n+1. If these changes are not tracked, renamed codes are often considered as new code. However, tracking these requires a lot of processing and is not always accurate. To avoid this problems, in this research, major renamings are identified manually and changed to a common name. For example, in Commons-lang 3.0, a package is named as 'lang', which is named as 'lang3' in all the next versions. As a result, all the classes and methods in 'lang3' is considered as new. To tackle this problem, this package is renamed to 'lang3' in all versions manually. There might exist some other smaller renamings which are not handled in this study.

6.2 Future Direction

This is the first work, investigating evolution of smell clusters to understand behavior of code smells. Therefore, the are many opportunities to improve on it, to extend it or to use it in a new direction.

Improvement. As an enhancement, this framework should be improved so that it can track classes and methods from version to version. This will help to reduce problems regarding project restructuring and renaming. This study currently only

considers four smells (God Class, Long Method, Feature Envy, Type Checking). To have more generalized results, other smells should also be considered in future researches. Cluster evolution of individual smells should be performed to understand more detail about evolution patterns.

Extension As it is seen in the literature, refactoring all the smells is not feasible. Therefore the future direction lies in development of a refactoring recommendation technique that will suggest minimum amount of refactorings to eliminate maximum possible smell relations. It will help to control the size of mega clusters, or even eliminate those.

New Direction. Another future direction is to develop a commit to commit smell tracking technique using the current concept (version to version). This technique will help to track smell clusters in real-time. As a result, prediction future smell clusters and accurate refactoring suggestions can be obtained. However, to work in this direction, a custom smell detection technique must be developed that can identify smells by comparing commit history.

There are many other opportunities to work on from this study. In future, the contributors of this research, intend to work on the above discussed directions to aid developers produce better code.

Appendix A

Reading List

Bibliography

[1] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards assessing software architecture quality by exploiting code smell relations," in *Proceedings of the Second International Workshop on Software Architecture and Metrics*, pp. 1–7, IEEE Press, 2015.

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

[3] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 411–416, IEEE, 2012.

[4] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pp. 255–258, IEEE, 2009.

[5] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Pearson Education India, 1999.

[6] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger, "Codecrawler-an information visualization tool for program comprehension," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 672–673, IEEE, 2005.

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

[8] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," in *Extreme Programming and Agile Processes in Software Engineering*, pp. 75–84, Springer, 2006.

[9] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, no. 2, pp. 127–139, 2003.

[10] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

[11] A. Lozano, K. Mens, and J. Portugal, "Analyzing code evolution to uncover relations," in *Patterns Promotion and Anti-patterns Prevention (PPAP), 2015 IEEE 2nd Workshop on*, pp. 1–4, IEEE, 2015.

[12] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[13] D. L. Parnas, "Software aging," in *Proceedings of the 16th international conference on Software engineering*, pp. 279–287, IEEE Computer Society Press, 1994.

[14] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proceedings of the 2nd international conference on Software engineering*, pp. 592–605, IEEE Computer Society Press, 1976.

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

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

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.

[18] R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, pp. 151–170, 1994.

[19] B. F. Webster, *Pitfalls of object oriented development*. M & T Books, 1995.

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

[21] K. Beck, M. Fowler, and G. Beck, "Bad smells in code," *Refactoring: Improving the design of existing code*, pp. 75–88, 1999.

[22] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 381–384, IEEE, 2003.

[23] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *ACM Sigplan Notices*, vol. 34, pp. 47–56, ACM, 1999.

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

[25] F. Khomh, M. D. Penta, and Y.-G. Gueheneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 75–84, IEEE, 2009.

[26] W. C. Wake, *Refactoring workbook*. Addison-Wesley Professional, 2004.

[27] R. Marticorena, C. López, and Y. Crespo, "Extending a taxonomy of bad code smells with metrics," in *7th ECCOP International Workshop on Object-Oriented Reengineering (WOOR)*, p. 6, 2006.

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

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

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

[31] R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc, "Numerical signatures of antipatterns: An approach based on b-splines," in *Software maintenance and reengineering (CSMR), 2010 14th European Conference on*, pp. 248–251, IEEE, 2010.

[32] P. Oliveira, M. T. Valente, and F. Paim Lima, "Extracting relative thresholds for source code metrics," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pp. 254–263, IEEE, 2014.

[33] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–10, IEEE, 2010.

[34] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*, pp. 305–314, IEEE, 2009.

[35] R. L. Nord, I. Ozkaya, H. Koziolek, and P. Avgeriou, "Quantifying software architecture quality report on the first international workshop on software architecture metrics," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 32–34, 2014.

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

[37] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1037–1039, ACM, 2011.

[38] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells.," in *ICSM*, pp. 519–520, 2007.

[39] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 329–331, IEEE, 2008.

[40] K. Dhambri, H. Sahraoui, and P. Poulin, "Visual detection of design anomalies," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pp. 279–283, IEEE, 2008.

[41] G. Langelier, H. Sahraoui, and P. Poulin, "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 214–223, ACM, 2005.

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

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

[44] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?," in *IASTED Conf. on Software Engineering*, pp. 346–355, 2006.

[45] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 682–691, IEEE Press, 2013.

[46] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 385–394, ACM, 2007.

[47] M. D. Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the relationships between design pattern roles and class change proneness," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 217–226, IEEE, 2008.

[48] F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "Playing roles in design patterns: An empirical descriptive and analytic study," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 83–92, IEEE, 2009.

[49] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of fault-proneness in methods by focusing on their comment lines," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 2, pp. 51–56, IEEE, 2014.

[50] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, *et al.*, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[51] A. Vetro, N. Zazworka, F. Shull, C. Seaman, and M. A. Shaw, "Investigating automatic static analysis results to identify quality problems: An inductive study," in *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*, pp. 21–31, IEEE, 2012.

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

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

[54] S. Jancke and D. Speicher, "Smell detection in context," *University of Bonn*, 2010.

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

[56] "JUnit - a simple framework to write repeatable tests." `http://junit.org/junit4/`. Accessed: 2016-04-16.

[57] "Apache commons-lang." `https://commons.apache.org/proper/commons-lang/`. Accessed: 2016-04-16.

[58] "Mockito - mocking framework for unit tests written in java." `http://mockito.org/`. Accessed: 2016-04-16.

[59] "GitHubs 10,000 most Popular Java Projects - here are the top libraries they use." `http://blog.takipi.com/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/` Accessed: 2016-04-16.

[60] "JUnit - github repository." `https://github.com/junit-team/junit4`. Accessed: 2016-04-16.

[61] "Mockito - github repository." `https://github.com/mockito/mockito`. Accessed: 2016-04-16.

[62] "Ccommons-lang - github repository." `https://github.com/apache/commons-lang`. Accessed: 2016-04-16.

[63] "Replication Package - input data of the case study, source code of 'smell cluster inspecor toolkit' and jar file of the tool." `https://www.dropbox.com/sh/c1umno5lk9j2wui/AADGVmYxeFajp6QsFsbtoRDNa?dl=0`. Accessed: 2016-04-16.